Marcos Vinicius Linhares

MODELO DE PROGRAMAÇÃO E SUPORTE DE EXECUÇÃO PARA APLICAÇÕES MULTITAREFA EM PROCESSADORES DSP DE PEQUENO PORTE

FLORIANÓPOLIS
2004

UNIVERSIDADE FEDERAL DE SANTA CATARINA

CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

MODELO DE PROGRAMAÇÃO E SUPORTE DE EXECUÇÃO PARA APLICAÇÕES MULTITAREFA EM PROCESSADORES DSP DE PEQUENO PORTE

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

Marcos Vinicius Linhares

Florianópolis, Março de 2004.

MODELO DE PROGRAMAÇÃO E SUPORTE DE EXECUÇÃO PARA APLICAÇÕES MULTITAREFA EM PROCESSADORES DSP DE PEQUENO PORTE

Marcos Vinicius Linhares

‘Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica, Área de Concentração em *Controle, Automação e Informática Industrial*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.’

Prof. Rômulo Silva de Oliveira, Dr.
Orientador

Prof. Daniel Juan Pagano, Dr.
Co-Orientador

Prof. Jefferson Luiz Brum Marques, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Prof. Antonio Heronaldo de Sousa, Dr.

Prof. Hari Bruno Mohr, Dr.

Prof. Luis Fernando Friedrich, Dr.

“Faço o melhor que sei, o melhor que posso e faço até o final.
Se ao fim, tudo deu certo, o que dizem não faz a menor diferença".
(Abraham Lincoln)

Dedico este trabalho à minha esposa, minha mãe e meu padrasto,
pelo amor, carinho, paciência e recursos doados.

AGRADECIMENTOS

À minha esposa, Bianca de Souza, por ter aceitado, algumas vezes não, as minhas variações de humor e sempre ter me apoiado.

À minha família que, sempre que possível, esteve presente durante as minhas realizações, acadêmicas ou não.

Ao meu orientador, Prof. Rômulo Silva de Oliveira, e ao meu co-orientador, Prof. Daniel Juan Pagano, pelo apoio e incentivo dado ao trabalho que se mostrou, no início, um terreno obscuro mas que com o tempo descortinou-se como uma área carente e ainda pouco explorada.

Aos meus colegas de Laboratório, com os quais passei incontáveis momentos de lazer (parada para o cafezinho, almoços no RU e várias trilhas pela ilha de Florianópolis), essenciais para dar continuidade ao desenvolvimento do trabalho.

Ao PPGEEL, em especial ao Wilson, ao Marcelo e aos professores Edson e Jefferson.

Aos amigos e professores que durante a graduação estiveram junto comigo durante a iniciação ao mundo científi co e acadêmico.

À Texas Instruments e à Grameyer que viabilizou os equipamentos utilizados durante a implementação do trabalho.

À minha mãe e ao CNPq pelo apoio fi nanceiro.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

MODELO DE PROGRAMAÇÃO E SUPORTE DE EXECUÇÃO PARA APLICAÇÕES MULTITAREFA EM PROCESSADORES DSP DE PEQUENO PORTE

Marcos Vinicius Linhares

Março/2004

Orientador: Prof. Rômulo Silva de Oliveira, Dr.
Co-Orientador: Prof. Daniel Juan Pagano, Dr.
Área de Concentração: Controle, Automação e Informática Industrial
Palavras-chave: Multitarefa, Sistemas Operacionais, DSP
Número de Páginas: xv + 118

Muitas das aplicações de controle com restrições de tempo são implementadas pela codificação de programas complexos em linguagem *assembly*, programando *timers*, manipulando em baixo nível dispositivos periféricos, tarefas e prioridades de interrupção. A maior conseqüência desta abordagem é que o *software* de controle é produzido por técnicas *ad hoc* que podem apresentar baixa confiabilidade e alto custo. Para o projeto e implementação de um sistema de controle de tempo real embutido se utiliza, basicamente, de dois tipos de profissionais, o engenheiro de controle e o engenheiro de *software*, que são especialistas cada um em sua área e que usam suas próprias ferramentas. Para tanto é de extrema valia que ambos troquem informações utilizando uma mesma linguagem. A ferramenta de projeto do engenheiro de controle são os diagramas de blocos funcionais que, mapeados em diagramas de componentes, fornecem ao engenheiro de *software* as informações necessárias. A simples utilização de blocos funcionais gera um sistema monotarefa, mas pressões do mercado exigem equipamentos que ofereçam maiores funcionalidades a seus clientes e isto representa mais tarefas a embutir no sistema. O modelo monotarefa começa a ser insuficiente, neste caso, para manter as restrições temporais no sistema final. Este trabalho descreve uma proposta que visa melhorar o processo de construção (projeto e implementação) de sistemas embutidos utilizando o modelo multitarefa. Para tanto, o trabalho consiste, principalmente, da proposta de um *framework* multitarefa e o desenvolvimento de um núcleo operacional de tempo real para dar suporte ao *framework* desenvolvido. Espera-se com este trabalho contribuir para a melhoria dos processos de desenvolvimento de *software* para sistemas embutidos no contexto do controle e automação.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

PROGRAMMING MODEL AND EXECUTION SUPORT FOR MULTITASKING APPLICATIONS ON SMALL DSP PROCESSORS

Marcos Vinicius Linhares

March/2004

Advisor: Prof. Rômulo Silva de Oliveira, Dr.
Co-Advisor: Prof. Daniel Juan Pagano, Dr.
Area of Concentration: Control, Automation and Industrial Computing
Key words: Multitasking, Operational Systems, DSP
Number of Pages: xv + 118

Often, many control applications with time restrictions are implemented by the codification of complex programs in assembly language, programming timers, manipulating peripheral devices, tasks and interruption priorities in low level. The consequence of this approach is that the control software is produced by ad hoc techniques that can present low reliability and high cost. The design and implementation of an embedded real-time control system uses, basically, two types of professionals: the control engineer and the software engineer, who are specialists on their areas and tools. It is of extreme value that both exchange information using the same language. The main tool used by the control engineer is the functional blocks diagram that, mapped to diagrams of components, supply the software engineer with the necessary information. The simple use of functional blocks generates a monotasking system, but pressures of the market demand equipment that offers greater functionalities for the customers and this represents more tasks to embed in the system. The monotasking model starts to be insufficient, in this case, to meet temporal restrictions in the final system. This work describes a proposal that aims to improve the process of construction (design and implementation) of embedded systems using the multitasking model. In this sense, the work consisted, mainly, of the multitasking framework proposal and the development of a real-time operational microkernel to support the developed framework. This work contributes to the improvement of software development processes for embedded systems in the context of control and automation.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Nos últimos anos tem-se visto um crescimento extraordinário da indústria eletrônica e um dos principais motivos está na incorporação de sistemas eletrônicos em uma grande variedade de produtos como automóveis, eletrodomésticos e outros. A introdução de sistemas eletrônicos em equipamentos tradicionais tornou-os mais eficientes, de melhor qualidade e mais baratos. Dentre estes componentes eletrônicos pode-se incluir aqueles que permitem algum tipo de computação: os microprocessadores e microcontroladores.

Estes componentes eletrônicos melhoraram bastante a qualidade do produto final mas o projeto dos sistemas tornaram-se bem mais complexos já que envolvem uma grande quantidade de elementos heterogêneos (componentes analógicos e digitais).

Existe uma tendência que os sinais analógicos sejam processados como sinais digitais, isso devido a grande facilidade de integração entre componentes digitais e a grande flexibilidade destes, pois podem envolver algum tipo de computação e controle em seu interior.

1.1 Conceitos básicos e justificativa

Diversos componentes digitais unidos com um determinado objetivo formam um sistema digital. Um sistema digital pode ser dividido em três classes básicas[6]:

- Sistemas de emulação e prototipação - baseados em tecnologias de *hardware* reprogramáveis, onde o *hardware* pode ser reconfigurado pelo uso de ferramentas específicas por usuários especialistas. Como exemplo tem-se o FPGA (*Field Programmable Gate Arrays*), um dispositivo de *hardware* que permite a programação de *gates* (circuitos eletrônicos simples, capazes de realizar operações lógicas e aritméticas, por exemplo, fundamentais para circuitos digitais);

- Sistemas de computação de propósito geral - desta classe fazem parte os computadores tradicionais, são caracterizados pela possibilidade do usuário final poder programar o sistema;
- Sistemas de computação e controle dedicados - caracterizados por serem desenvolvidos para uma aplicação específica, onde o usuário final tem acesso limitado à programação do sistema.

Os sistemas de computação podem ser utilizados para controlar uma grande variedade de equipamentos, desde simples máquinas domésticas até completas instalações de produção. Quando estes sistemas estão firmemente fixados aos equipamentos ao ponto de que, se forem retirados, o equipamento deixa de funcionar então estes sistemas são chamados de sistemas embutidos (*embedded systems*)[1][22].

Normalmente, estes sistemas possuem restrições temporais na execução das tarefas para as quais foram projetados e programados e quando isto acontece tem-se um sistema de tempo real.

Os sistemas de tempo real são sistemas cujo funcionamento correto depende dos resultados produzidos por ele e do instante no qual esses resultados são produzidos. Um sistema de tempo real brando (*soft*) é aquele cuja operação será degradada se os resultados não forem produzidos de acordo com os requisitos de tempo especificados. Um sistema de tempo real é crítico (*hard*) quando os resultados produzidos por ele são incorretos se não respeitam os requisitos de tempo, podendo resultar em catástrofes [15][16][24]. Exemplos de aplicações que requerem tempo real incluem:

- Controle de uma planta química e nuclear;
- Controle de processos complexos de produção;
- Aplicações automotivas e aeronáuticas;
- Sistemas de telecomunicações;
- Automação industrial e robótica;
- Sistemas militares e muitos outros.

Retirando este vasto domínio de aplicações, muitos pesquisadores e desenvolvedores possuem várias concepções sobre computação de tempo real[25] e muitos dos atuais sistemas de tempo real são projetados utilizando técnicas e abordagens empíricas, através de conhecimentos adquiridos por tentativa e erro. Muitas das aplicações de controle com restrições de tempo são implementadas pela codificação de grandes programas em linguagem *assembly*, programando *timers*, manipulando em baixo nível dispositivos periféricos, tarefas e prioridades de interrupção. Apesar do código produzido por estas técnicas serem otimizados para uma execução eficiente, esta abordagem tem algumas desvantagens[4]:

- A implementação de grandes partes de código em linguagem *assembly* é muito mais difícil e consome muito mais tempo que a programação em linguagens de alto nível;
- Exceto para os programadores que desenvolveram a aplicação, poucas pessoas conseguem entender o funcionamento completo do *software* produzido. E muitas vezes programadores bastantes experientes inserem mais complexidade ao código devido ao conhecimento adquirido;
- Como a complexidade da programação aumenta, as modificações de grande parte do programa tornam-se muito difíceis, até mesmo para o programador original, o que leva muitas vezes a se jogar o código fora e se refazer o trabalho;
- Sem o suporte de ferramentas e metodologias específicas, análises para verificação de restrições temporais são praticamente impossíveis.

A maior conseqüência deste abordagem é que o *software* de controle produzido por técnicas empíricas podem ser imprevisíveis. Se todas as restrições de tempo não forem verificadas anteriormente e o sistema operacional, quando existente, não incluir características para manipular tarefas de tempo real, o sistema pode aparentemente funcionar corretamente por um período de tempo mas pode entrar em colapso em situações não previstas. As conseqüências da falha do sistema dependem da criticidade da aplicação.

Para o projeto e implementação de um sistema de controle de tempo real embutido se utiliza de diversos tipos de profissionais que são especialistas cada um em sua área e que usam suas próprias ferramentas. Um sistema simples envolveria um engenheiro de *hardware* para projetar o *hardware* a ser utilizado, um engenheiro de controle para projetar o controlador e impor as restrições temporais e um engenheiro de *software* para implementar o controlador no *hardware* construído.

Pode-se perceber que a interação entre o engenheiro de controle e o de *software* deve ser a mais eficaz possível já que um está projetando o que o outro irá implementar. E para tanto é de extrema valia que ambos troquem informações utilizando uma mesma linguagem. A ferramenta de projeto do engenheiro de controle são os diagramas de blocos funcionais onde estão contidos os modelos matemáticos da planta onde se está atuando e do controlador em questão. De forma a facilitar a comunicação entre eles pode-se mapear os diagramas de blocos em diagramas de componentes de *software* onde o engenheiro de *software* pode retirar as informações para implementar o controlador.

A simples utilização de blocos funcionais gera um sistema monotarefa onde o objetivo deste é controlar um determinado equipamento ou processo e nada mais, respeitando os requisitos temporais, recaindo-se invariavelmente em uma codificação de baixo nível (programação *assembly*) com grande dificuldade de manutenção. Mas pressões do mercado exigem equipamentos não só robustos mas que ofereçam maiores funcionalidades a seus clientes e isto representa para o engenheiro de *software* mais tarefas a embutir em um determinado dispositivo. O modelo monotarefa começa a ser insuficiente, neste caso, para manter as restrições temporais no sistema final.

1.2 Objetivos

O objetivo deste trabalho de mestrado é estender um *framework* monotarefa já existente, que utiliza o mapeamento de diagramas de blocos para componentes de *software*, para um modelo multitarefa, mantendo as características principais do *framework* original.

Para tanto, dois esforços principais serão feitos:

- A criação de um *framework* multitarefa que englobará o original e criará novas representações para o modelo multitarefa;
- E o desenvolvimento de um núcleo operacional de tempo real (μKernel) para dar suporte de execução ao *framework* desenvolvido.

Não é preocupação deste trabalho a otimização do μKernel para se obter um maior desempenho do sistema operacional de suporte. É preocupação deste trabalho possibilitar manter as características do projeto de sistemas utilizando diagramas de blocos e possibilitar a sua utilização em um ambiente multitarefa.

1.3 Organização

A organização desta dissertação reflete uma crescente evolução no conhecimento adquirido pelo mestrando. Desde conceitos básicos de automação até a implementação de um sistema operacional de tempo real para aplicações utilizando controladores embutidos.

No capítulo 2 é dado um panorama de aplicações dos sistemas de automação e os diversos níveis hierárquicos nos quais pode estar presente. São inseridos conceitos sobre técnicas e ferramentas utilizadas na concepção de sistemas de automação com auxílio de computador. Para encerrar são inseridos alguns conceitos sobre controladores embutidos e seus aspectos temporais.

No capítulo 3 é apresentado o *meta-framework* proposto pela Texas Instruments para o desenvolvimento de sistemas embutidos em processadores digitais de sinais (DSP). Abrange desde técnicas de programação (*Algorithm Standard*), ferramentas de *software* e um *framework* para projeto de sistemas utilizando diagramas de blocos, juntamente com uma aplicação exemplo do *framework*

O capítulo 4 apresenta o *framework* multitarefa proposto onde são introduzidas novas representações a serem incorporadas com o objetivo de possibilitar o projeto de sistemas multitarefa. Insere, além de símbolos para delimitar as tarefas, tipos de dados abstratos para permitir a comunicação entre elas.

No capítulo 5 é descrito o projeto, API (*Application Protocol Interface*) e codificação do núcleo operacional de tempo real para suportar o *framework* multitarefa proposto. São descritos seus mecanismos e estruturas internas, concluindo com uma tabela que demonstra a quantidade e tipo de memória utilizada pelo núcleo. O código fonte do μKernel consta como apêndice deste trabalho.

O capítulo 6 aplica o *framework* proposto juntamente com o μKernel desenvolvido em uma aplicação que será executada em um processador digital de sinais (DSP) da Texas Instruments. É mostrado passo a passo a concepção da aplicação desde a engenharia de requisitos até o projeto e codificação, finalizando com o comportamento temporal da aplicação exemplo.

Finalmente, o capítulo 7 contém algumas considerações sobre o trabalho desenvolvido e apresenta os trabalhos futuros que poderão ser desenvolvidos a partir deste.

Capítulo 2

Automação e Controladores Embutidos

O público alvo deste trabalho abrange tanto os profi ssionais de engenharia de controle como os profi ssionais de computação. Portanto, tentou-se com este capítulo contribuir com uma visão geral da área de automação e controle para os profi ssionais da computação não esquecendo dos aspectos temporais dos controladores para os engenheiros de controle.

2.1 Introdução

Controle e automação são termos da língua portuguesa, uma linguagem informal. Logo, é razoável esperar um certo grau de ambigüidade, incompletude e incoerência na defi nição destes termos. Do dicionário[7]: controle - “ato ou poder de controlar, domínio, governo”; automação é - “sistema pelo qual os mecanismos controlam seu próprio funcionamento, quase sem interferência do homem".

Pode-se dizer que a idéia de automação (mecanismos que controlam seu próprio funcionamento) é realizada através da idéia de controle, onde o controlador não é humano. Na prática a maioria dos sistemas de automação incluem, além de mecanismos de controle, funcionalidades tais como interface humano-computador e integração com sistemas de informação administrativos.

Caso se entenda que o controle implica somente em uma ação cega, sem correção, tem-se um sistema aberto (geralmente denominado de ‘malha aberta’). Se este controle leva em consideração as informações passadas e calcula a ação corretiva mais apropriada tem-se um sistema fechado (‘malha fechada’)[21].

Na ‘malha fechada’, prevê-se o uso extensivo dos mesmos conceitos da ‘malha aberta’. Entretanto, o nível de flexibilidade imputado ao sistema é bem mais elevado, pelo fato de possuir o conceito de realimentação associado, este sistema é também denominado como laço de controle. Esta propriedade possibilita, a um sistema dotado de realimentação, modifi car o comportamento automatizado com a fi nalidade de, intencionalmente, produzir resultados diferenciados.

É importante notar que automação não significa necessariamente automação industrial. Embora a automação dos processos industriais seja o maior segmento, existem diversas outras classes de aplicações onde esse conceito é utilizado:

- Automação predial (iluminação, climatização, acesso, hidráulica, etc);
- Automação veicular (ignição eletrônica, freios ABS, ar-condicionado, etc);
- Automação de tráfego (semáforos, onda verde, etc);
- Automação doméstica (geladeira, freezer, microondas, etc);
- Automação no escritório (scanner, fax, impressora, telefonia, etc);
- Automação no entretenimento (vídeo-game, DVD-player, CD-player, etc).

Embora, no passado, fossem utilizados mecanismos totalmente eletro-mecânicos na construção de sistemas de automação, atualmente, a computação está presente em praticamente todos os sistemas dotados de automação. A implementação destes sistemas e de suas soluções de controle automático está fundamentada no uso de *hardware* e *software*, onde a computação tornou-se elemento vital e central para a automação.

O computador passou a ser parte integrante dos sistemas de automação e a interconexão entre eles permitiu que a automação penetrasse definitivamente em todos os setores da indústria, aumentando a produtividade, a qualidade dos produtos e a flexibilidade da produção.

2.2 O Modelo Hierárquico da Automação

Uma planta industrial deve continuamente ajustar a produção para atender as necessidades do mercado, mantendo uma alta produtividade e qualidade, e os custos de produção os mais baixos possíveis. As decisões estratégicas, provenientes de níveis superiores devem, de alguma forma, ser consideradas e incorporadas ao processo produtivo [10][18].

Para tanto divide-se a planta industrial em níveis hierárquicos de forma a tornar eficiente a comunicação entre componentes de um mesmo nível, horizontalizando e uniformizando os detalhes técnicos e verticalizando a informação, ou seja, níveis hierárquicos superiores não devem se preocupar com detalhes de implementação dos níveis inferiores, mas devem ter acesso às informações necessárias à tomada de decisões. A Figura 2.1 ilustra a estrutura de divisão em níveis hierárquicos.

A divisão dos sistemas em vários níveis permite que cada nível seja analisado e projetado separadamente, particionando a complexidade do problema como um todo. A quantidade de níveis

utilizados e a divisão entre eles depende do tipo e do tamanho do sistema analisado. De maneira a simplificar adotou-se uma estrutura mais genérica (Figura 2.1) descrevendo-se os níveis, que de acordo com o sistema em consideração podem ser subtraídos ou mesclados a outros níveis.

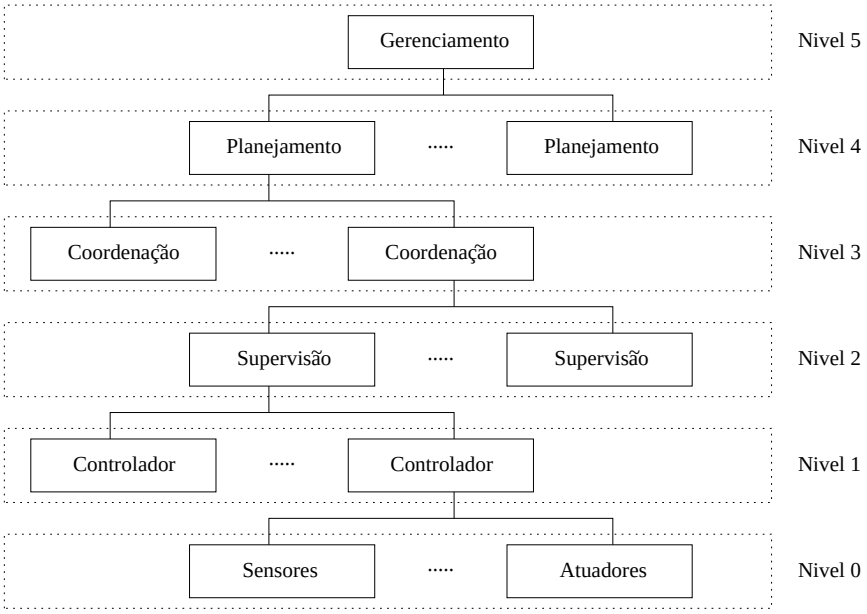

Figura 2.1: Estrutura de Níveis Hierárquicos.

- Nível 5 - fornece informações sobre o estado da planta, da empresa e da produção para a área administrativa (compras, vendas, contabilidade, etc) que é responsável pelas decisões estratégicas da empresa.
- Nível 4 - planejamento de produção de um produto, atividades relacionadas com o gerenciamento operacional, isto é, planejar o uso eficiente de recursos, minimizar custos, maximizar produção e qualidade dos produtos, minimizar os atrasos no atendimento dos pedidos. Este nível é responsável, também, por projetar ou definir o produto, o processo e as necessidades de fabricação.
- Nível 3 - adapta e coordena o plano de produção, gerado pelo nível superior, em um subsistema local de modo a otimizar o desempenho e possibilitar uma alocação eficiente dos recursos locais.
- Nível 2 - controla o conjunto de unidades que definem uma sub-área de uma planta, dirige o funcionamento cooperativo e correto dos componentes do nível inferior. Emergências ocorridas nestes níveis (nível 2 e nível 1) devem iniciar ações corretivas para manter a segurança e operacionalidade dos equipamentos.
- Nível 1 - responsável pelo controle operacional direto da planta, onde uma grande variedade de equipamentos (robôs, tornos, fresadoras, motores, etc), que utilizam controladores dedicados

baseados em microprocessadores, estão trocando informações de forma a possibilitar o efetivo funcionamento da estratégia de controle desenvolvida.

- Nível 0 - neste nível encontram-se os transdutores (sensores e atuadores), portadores de alguma inteligência local ou não, e outros equipamentos que cooperam, de forma distribuída, para que a estratégia de controle possa ser aplicada respeitando os tempos ou outra restrição qualquer imposta pelos níveis superiores.

Um sistema de automação de uma planta industrial envolve o processamento de um grande número de variáveis provenientes de sub-sistemas com amplo espectro dinâmico. Este tipo de sistema requer o desenvolvimento de muitas funções, algumas das quais extremamente complexas, de forma a transformar os valores das variáveis em sinais de controle corretivos. Os sinais de controle, por sua vez, devem ser transmitidos a uma grande variedade de mecanismos de atuação.

Muitas das aplicações de controle se mostram insatisfatórias com respeito aos requisitos temporais pois, dentro de um curto período de tempo, existe a necessidade de obter uma alta velocidade de resposta após a amostragem de um sinal a ser controlado. Basta um atraso na realimentação do sistema e os novos dados irão gerar uma solução de controle errada, baseada em dados desatualizados. O problema é tão maior quanto maior for o atraso [21]. Tais problemas existem e são geralmente encontrados em sistemas de automação com requisitos temporais restritos (tempo real). À medida que as atividades aproximam-se dos níveis mais baixos, da hierarquia, aumenta o rigor das restrições temporais das tarefas computacionais e a demanda por tempos de resposta mais curtos [20]. Portanto, os níveis inferiores (0 e 1) devem possuir autonomia e flexibilidade suficiente para poderem resolver seus problemas sem a necessidade que um nível superior interceda por eles. A implementação de controladores nos níveis inferiores será discutida mais adiante.

2.3 Controle Auxiliado por Computador

O computador vem sendo introduzido nos mais diversos segmentos da indústria, estando presente desde o nível do processo, ao nível de gestão e administração da empresa, resultando em um conjunto variado de atividades executadas com o *auxílio do computador*.

O grande avanço nos processadores, memórias e outras 'infra-estruturas' de *hardware* e *software* têm mudado profundamente o modo de viver e trabalhar. Estes avanços têm possibilitado o surgimento de novos níveis de funcionalidade e inteligência em sistemas de controle, em particular, a infra-estrutura computacional possibilitou realizar cálculos complexos no desenvolvimento de muitas teorias de controle.

Nas últimas quatro décadas os computadores vêm ganhando grande importância nos sistemas de controle. Programas de computador para controle automático tem sido construídos, tradicionalmente,

para duas diferentes propostas: ajudar o Engenheiro de Controle na modelagem, análise e projeto de sistemas de controle; e implementar programas para a concepção de sistemas reais de controle [19]. Os principais estágios envolvidos no ciclo de desenvolvimento de controladores são mostrados na Figura 2.2.

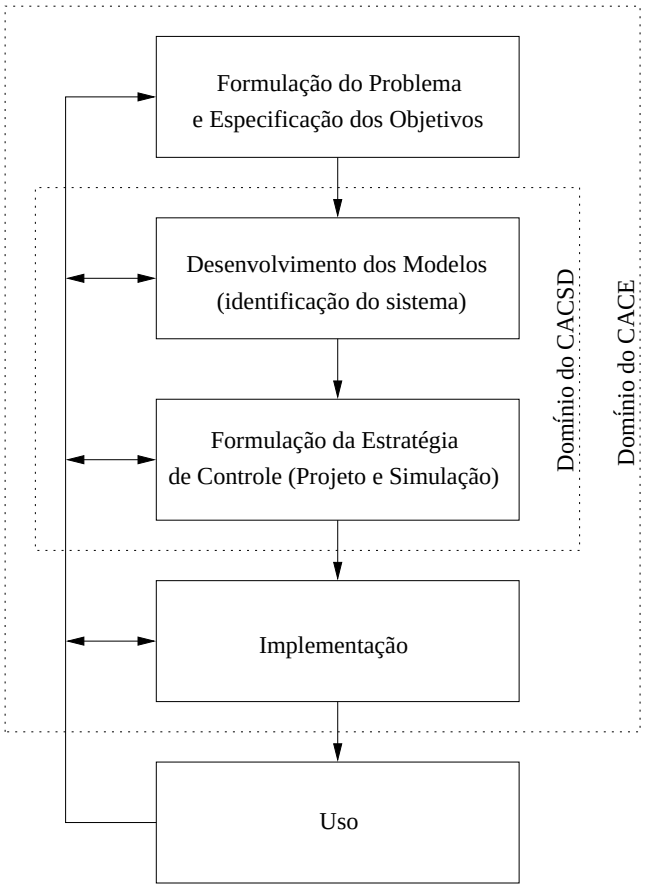

Figura 2.2: Ciclo de Desenvolvimento de Controladores.

A tarefa de desenvolvimento dos controladores segue uma estrutura organizacional hierárquica dividida em estágios para possibilitar uma melhor orientação ao projetista de controladores.

O primeiro estágio no ciclo de desenvolvimento de controladores é a fase conceitual de formulação do problema e especifi cação dos objetivos de controle (e.g., regular um equipamento para minimizar o consumo de energia, controlar o movimento mecânico de um braço de robô, etc).

Após os objetivos de controle terem sido formulados, um modelo dinâmico do processo é desenvolvido. Esta atividade inclui a utilização das mais diversas técnicas de projeto desde as puramente empíricas (ditadas pela experiência adquirida pelo engenheiro) até aquelas fortemente baseadas em teorias formais de análise e síntese de sistemas.

O próximo passo no projeto de controladores é desenvolver uma estratégia de controle que irá atingir os objetivos de controle. A simulação por computador é uma importante técnica na modelagem

e avaliação das alternativas de estratégias de controle. Ressalta-se que a utilização de protótipos ou de modelos experimentais é uma possibilidade freqüentemente utilizada para o suporte de projetos com alguma complexidade e para o teste real da estratégia de controle utilizada.

Na implementação, o *hardware* é selecionado de acordo com as restrições impostas (custos, disponibilidade) e o *software* do sistema de controle é construído. Esta etapa freqüentemente origina modificações no projeto original (pois a tecnologia escolhida pode possuir restrições não previstas) e, quando isto ocorre, algumas fases da etapa anterior devem ser refeitas.

Finalmente, pronto o sistema de controle (*hardware* e *software*) este é instalado na planta. Então os controladores são ajustados na planta real usando como ponto inicial as estimativas verificadas nos passos de projeto. Normalmente esse ajuste é um procedimento de tentativa e erro.

Dependendo da experiência do projetista e do sistema que está sendo considerado, alguns dos passos no ciclo de desenvolvimento podem ser omitidos. Quanto mais iterações no ciclo melhor o processo de desenvolvimento do controlador e o próprio controlador desenvolvido.

Existe um grande número de ferramentas para ajudar o engenheiro de controle nas diferentes fases do ciclo de desenvolvimento, essas ferramentas fazem parte de um universo chamado CAE (*Computer-Aided Engineering*) onde estão incluídas todas as ferramentas de auxílio à engenharia. Um termo que ganhou amplo uso, somente a poucos anos, é o Projeto de Sistemas de Controle Auxiliado por Computador (*Computer-Aided Control System Design* - CACSD). Uma definição mais formal do termo é dado por Jamshidi e Herget [14]:

> *"O uso de computadores digitais como ferramenta primária durante as fases de modelagem, identificação, análise e projeto da Engenharia de Controle."*

O CACSD inclui ferramentas de *software* para todas as fases do ciclo de projeto exceto para a implementação do controlador. Recentemente o termo Engenharia de Controle Auxiliado por Computador (*Computer-Aided Control Engineering* - CACE) tem se tornado popular. CACE é uma forma especializada de CAE e é definido como [19]:

> *"Uma forma especializada de modelagem e simulação com ênfase no projeto e implementação de sistemas de controle realimentados."*

O uso de ferramentas de *software* durante a formulação do problema não é muito comum, a principal razão é que as decisões feitas durante esta fase são baseadas, principalmente, em operações não numéricas e normalmente empíricas (dependem do conhecimento adquirido pelo engenheiro em experiências anteriores).

Já durante o estágio de implementação existe a dificuldade de padronização das linguagens de programação utilizadas e na grande quantidade de plataformas de *hardware* disponíveis cada uma com recursos que diferem entre equipamentos muitas vezes do próprio fabricante. Se a plataforma alvo não impõe restrições de custo, memória, tempo e outras, pode-se utilizar linguagens de alto nível, orientadas a objeto, mas a medida que as restrições impostas aumentam, a primeira característica das linguagens de alto nível que, geralmente, se perde é a flexibilidade pois precisam ser substituídas por linguagens de baixo nível (*Assembly*) e que muitas vezes totalmente diferentes de uma plataforma para outra.

Para os outros estágios (modelagem, simulação e projeto) várias têm sido as ferramentas desenvolvidas. O desenvolvimento de ambientes CACE (que integram todos os estágios) têm levantado muitas questões e originando vários problemas que são comuns a muitos outros ambientes no domínio do CAE, como: projeto de circuitos, engenharia de *software*, engenharia da manufatura, etc.

2.4 Controladores Embutidos

Um aspecto importante da automação é o desenvolvimento de equipamentos específicos cujo controlador microprocessado está embutido no mesmo. Tais equipamentos serão usados como componentes mais tarde no momento da integração do sistema. Em geral, os requisitos de controle do equipamento implicam em requisitos de natureza funcional e temporal para o *software* e o *hardware* [9].

O desenvolvimento de um produto com controlador embutido apresenta facetas mecânicas, elétricas, computacionais, etc e é normalmente denominado de engenharia de sistemas [18]. Um processo simplificado para o desenvolvimento de equipamentos que incluem um computador embutido, responsável pelo seu controle, poderia ser:

1. Especificação do equipamento como um todo, aspectos funcionais e outros como desempenho, segurança, dimensões, aparência, etc. Este trabalho é chamado de engenharia de produto e tem uma participação importante do *marketing*.
2. Projeto eletro-mecânico, conduzido por engenheiros elétricos e/ou mecânicos.
3. Ao projeto eletro-mecânico deve ser incluída a instrumentação necessária, que será utilizada como parte do sistema de controle embutido.
4. A partir do projeto eletro-mecânico pode ser feito o projeto da estratégia de controle. Necessidades da estratégia de controle podem alterar a instrumentação necessária. Se for impossível para o engenheiro de controle atender as especificações do produto a partir do projeto eletro-mecânico feito, o mesmo terá que ser revisto.

5. A partir do projeto da estratégia de controle, inicia o projeto do *hardware* e do *software*. Esta etapa é chamada na literatura de co-projeto de *hardware* e *software* (*hardware/software co-design*)[27]. O primeiro passo é fazer o particionamento entre o que será feito por *software* e o que será feito por *hardware*. Aspectos como desempenho e custo são considerados.

6. Após o particionamento, *software* e *hardware* podem a princípio serem desenvolvidos em paralelo. O *hardware* pode usar microprocessador, FPGA (*Field-Programmable Gate Array*), ASIC (*Application-Specific Integrated Circuit*), etc. O *software* pode ser programado em Java, C, Assembly, etc.

7. Com o *hardware* e o *software* prontos é feita a integração do equipamento, colocando todas as partes juntas.

8. Testes de sistema são então realizados, podendo existir a necessidade de correções no projeto.

Pela descrição do processo de concepção de um equipamento com controlador embutido, podemos perceber que são envolvidos vários tipos de profissionais:

- Engenheiro de produto;
- Engenheiro mecânico;
- Engenheiro eletricista;
- Engenheiro de controle;
- Engenheiro de computação ou eletrônico para fazer o *hardware*;
- Engenheiro de *software* (informata) para fazer o *software*.

Cada profissional segue o seu próprio sub-processo de trabalho, adequado para a tarefa que ele desempenha. Apesar de haver muitos pontos de interação entre eles, cada profissional possui seu próprio processo de desenvolvimento, o qual inclui atividades, métodos, linguagens, ferramentas e produtos. No caso dos profissionais de computação, eles recebem como especificação a estratégia de controle criada pelo engenheiro de controle e possivelmente algumas funcionalidades adicionais que, embora realizadas através do *software* e *hardware* embutido, não fazem parte de uma estratégia de controle automática.

Por exemplo, o equipamento em questão pode oferecer uma interface de usuário simples, através de teclado e *display*, onde uma pessoa poderá digitar comandos diretamente no equipamento. Ou ainda, o equipamento poderá ser conectado em rede local usando um protocolo específico, o qual deve ser implementado pelo sistema computacional embutido. Tais funcionalidades, que estão além da estratégia de controle embutida, passam diretamente da especificação do produto para os profissionais de computação.

2.5 Aspectos Temporais dos Controladores

O desenvolvimento sistemático de sistemas de controle com restrições temporais requer uma arquitetura de sistemas e metodologia de projeto apropriada. Sistemas computacionais de tempo real vêm substituindo cada vez mais os convencionais sistemas de controle mecânicos ou hidráulicos nas mais variadas aplicações como, por exemplo, nos controladores de vôo das aeronaves e sistemas de controle de processos em uma indústria. Em adição às capacidades funcionais específicas, estas aplicações passam a necessitar de atributos não-funcionais como, previsibilidade, segurança e manutenabilidade[4].

Atualmente, o processo de projeto de um sistema de tempo real é tedioso e muitas vezes não possui nenhuma metodologia. Freqüentemente o foco principal durante o projeto está sobre as capacidades funcionais do sistema de controle planejado. Considerações sobre tempos de resposta e dependências do sistema são deixadas de lado até a parte final de testes, quando todo o sistema é integrado.

Grande parte dos sistemas de controle de tempo real são sistemas embutidos onde o computador é um componente de uma vasto sistema de engenharia. O sistema de controle é muitas vezes implementado em um microcontrolador com um núcleo operacional de tempo real. Os núcleos de tempo real utilizam multiprogramação para executarem várias tarefas e devem poder garantir as restrições de tempo de cada tarefa do sistema. Durante os últimos anos o escalonamento de tarefas de tempo real tem sido uma área bastante pesquisada e um grande número de diferentes tipos de modelos e métodos de escalonamento tem sido desenvolvidos.

O modelo mais comum, e simplificado, utilizado dentro da comunidade de escalonamento tempo real assume que as tarefas são ou podem ser transformadas em tarefas periódicas com períodos fixos, um pior caso para o tempo de execução e um tempo de resposta que pode ser crítico (*hard*) ou brando (*soft*).

Um exemplo comum utilizado para ilustrar o escalonamento de tempo real quando o modelo é aplicado em sistema de controle computacional pode ser visto no laço de controle ilustrado na Figura 2.3, onde a cada período o controlador deve amostrar a saída do processo (y), executar o algoritmo de controle e enviar um novo sinal de controle (u) para a entrada do processo[5].

A performance e estabilidade do sistema depende, agora, não somente do algoritmo de controle mas também da implementação e comportamento em execução do controlador em um sistema computacional. O *hardware* utilizado, as propriedades de execução do algoritmo de controle, o sistema operacional de tempo real, o algoritmo de escalonamento, e outros, geram uma soma de atrasos que podem interferir no controle do sistema, influenciando na performance e estabilidade do controlador.

A proposta do escalonamento de tempo real é permitir que os requisitos de tempo de todas as tarefas sejam atendidos. Em um modelo de tarefa básico, ilustrado na Figura 2.4, uma tarefa é descrita

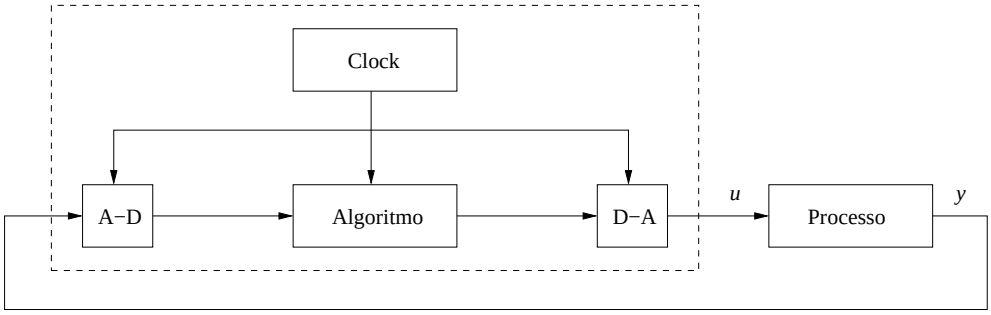

Figura 2.3: Uma tarefa de controle típica[5].

por um período (T), um *deadline* (tempo de resposta máximo - D), um tempo de resposta (R) e um tempo de computação (C).

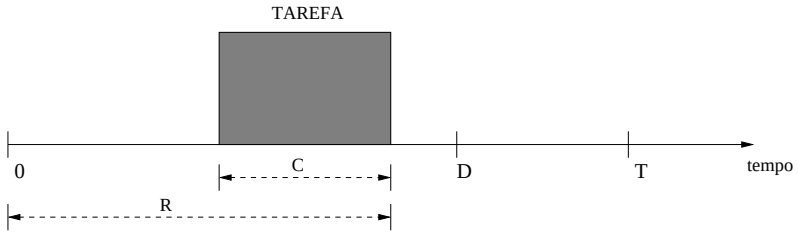

Figura 2.4: Modelo básico de uma tarefa tempo real[5].

As análises de escalonabilidade são utilizadas para prever, em tempo de projeto, se todas as tarefas cumprem seus requisitos temporais, isto é, se e somente se o tempo de resposta for menor ou igual ao seu *deadline* ($R \leq D$), para todas as tarefas. Cada algoritmo de escalonamento possui seus próprios métodos para análise de escalonabilidade.

Além do algoritmo de escalonamento os sistemas controle de tempo real devem possuir um suporte adequado para as aplicações e algumas características são importantes, como:

- Os resultados devem ser corretos, não somente no valor mas também no domínio do tempo. Como conseqüência o sistema operacional deve prover mecanismos específicos para gerenciamento do tempo e manipulação das tarefas com restrições explícitas de tempo para diferentes tipos de tarefas.
- Para garantir um nível mínimo de performance, o sistema deve ser capaz de prever as conseqüências de uma decisão de escalonamento. Se alguma tarefa não puder garantir seus requisitos de tempo, o sistema deve notificar o acontecimento e tomar as devidas ações alternativas.
- Falhas no *hardware* e/ou no *software* não devem levar o sistema ao colapso. Entretanto, componentes críticos de um sistema tempo real tem que ser projetados para ser tolerante a falhas.
- A arquitetura de um sistema de tempo real deve ser projetada de acordo com uma estrutura modular que permita que possíveis modificação no sistema possam ser feitas facilmente.

2.6 Conclusão

Pretendeu-se aqui dar uma panorâmica dos diversos sistemas de automação e sua aplicação nos mais diversos níveis hierárquicos desde os níveis gerenciais até os níveis de sensores e atuadores. O assunto é bastante vasto e envolve as mais diversas áreas.

Introduziu-se os conceitos de controladores e a utilização de sistemas computacionais para o auxílio desde o projeto até a implementação e suas problemáticas de integração.

Mostrou-se ainda processo de construção de controladores embutidos e a grande variedade de profi ssionais que esta área requisita. Finalmente apresentou-se os aspectos temporais dos controladores e suas restrições em aplicações com requisitos de tempo.

Chapter 3

Meta-Framework Proposto pela Texas Instruments

Neste capítulo serão descritos elementos para o desenvolvimento eficiente de sistemas embutidos com processadores digitais de sinais (*DSPs - Digital Sinal Processors*). Tomou-se como base um *meta-framework* elaborado pela Texas Instruments[1] (TI) devido a alguns fatores que se fizeram determinantes: é o principal fabricante mundial de *DSPs*; a existência, no departamento, de kits e ferramentas de desenvolvimento além da demanda existente no mercado catarinense atual onde várias empresas estão utilizando seus processadores e suas ferramentas.

3.1 Introdução

Os benefícios de um *software* estruturado em módulos é bem conhecido, especialmente, em sistemas complexos onde vários módulos contribuem individualmente na formação de todo o sistema. Os esforços de desenvolvimento dos módulos são realizados somente uma vez pois, os módulos podem ser reutilizados em projetos futuros. Para o engenheiro de controle esses módulos normalmente são conhecidos como blocos funcionais e devem possuir um comportamento previsível e serem compatíveis com outros blocos tanto no formato dos dados como funcionalmente, integrando o que os engenheiros chamam de diagrama de blocos (sistema). Para possuir estas características os algoritmos devem seguir um padrão, um conjunto de regras a serem utilizadas por todos os desenvolvedores.

As indústrias têm aprendido os benefícios desta metodologia e têm visto os impactos que esta pode causar sobre a integração de um sistema: tempo de depuração reduzido (pois os módulos já foram depurados exaustivamente, antes de serem liberados); rápida reconfiguração do sistema (os

[1] Mais informações podem ser encontradas em: www.ti.com

módulos são facilmente substituíveis por versões mais atualizadas ou retirados se estiverem causando algum problemas à aplicação); além de prover uma visualização de maior nível do software, encapsulando detalhes de implementação dos módulos.

A TI (*www.ti.com*) com o objetivo de aumentar a performance no desenvolvimento de sistemas utilizando *DSPs* criou um *meta-framework* para suportar a metodologia de desenvolvimento em módulos e a denominou *eXpressDSP*. O *eXpressDSP* é composto, basicamente, por elementos que têm como alvo o *DSP* e elementos que têm como alvo o PC. Entre os elementos que têm como alvo o *DSP* encontram-se o *TMS320 DSP Algorithm Standard*, o *DSP real-time kernel* e um *framework* de referência. Já entre as elementos que têm como alvo o PC estão o *Code Composer Studio* e o emulador *JTAG*. Na Figura 3.1 pode-se visualizar a divisão das diversas partes em seus respectivos alvos.

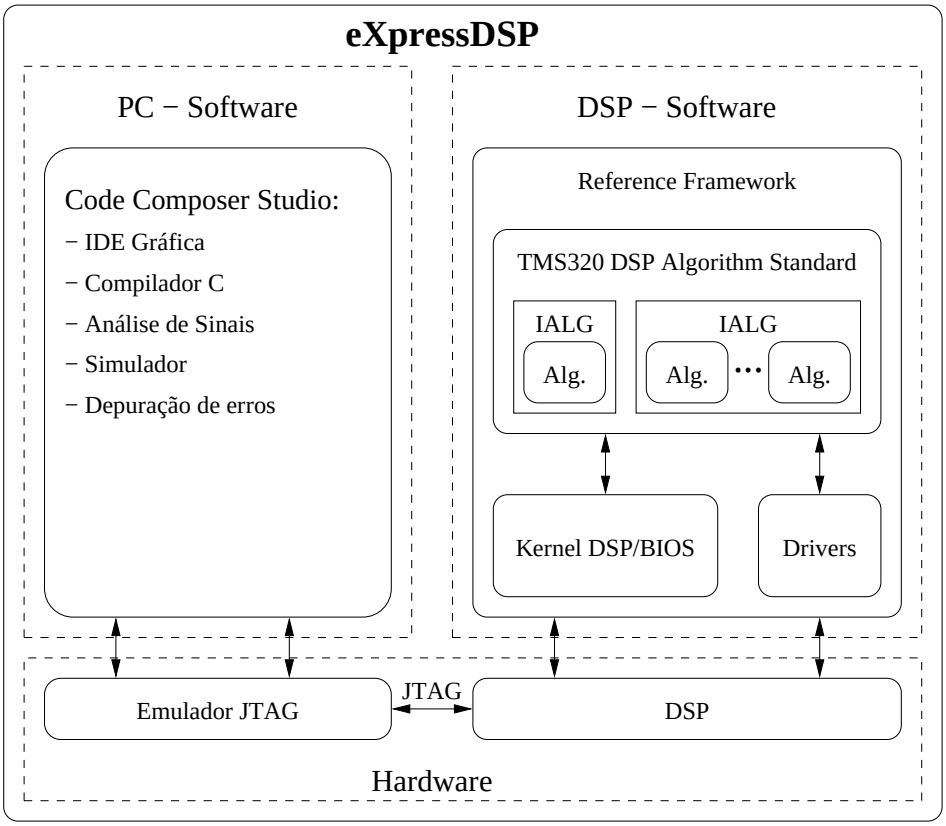

Figura 3.1: Elementos do *eXpressDSP*

- O *TMS320 DSP Algorithm Standard* (também conhecido como *XDAIS*) - especifica um conjunto de regras gerais e linhas guia (*guidelines*) a serem aplicadas ao desenvolvimento de algoritmos para *DSPs*. A proposta desta padronização é reduzir aqueles fatores que proíbem um algoritmo de ser facilmente integrado em um sistema, sem uma significante reengenharia de *software*;

- *Kernel DSP/BIOS* - camada de *software* de "baixo nível" que provê abstração de *hardware* e gerencia os recursos físicos. Também provê suporte a interrupções, *threads* e outras funções;
- *Reference Framework* - a "cola" que mantém juntos os *drivers*, os algoritmos, os gerenciadores de recursos e o *kernel DSP/BIOS*.
- *Code Composer Studio* - Ambiente integrado de desenvolvimento (IDE) que une as ferramentas de *software* necessárias para se trabalhar com *DSPs*, sendo exemplos delas: emulação, monitoração e controle em tempo de execução; configuração visual de periféricos e módulos do DSP/BIOS; compilação, linquedição e *download* do *software* para o DSP; e outros;
- Emulador *JTAG* (*Joint Test Action Group*) - *hardware* externo ligado à porta paralela ou ao barramento do PC e que permite emulação, por *software*, em tempo real, do *DSP* (ou *DSPs*), através do padrão IEEE 1149.1 que especifica instruções de varredura e teste e o *hardware* necessário para efetuá-lo.

Este *meta-framework*, proposto pela Texas Instruments, provê uma sólida integração do *software* com o *hardware* de desenvolvimento. Os benefícios de ser obediente à ele é o acesso à uma extensa rede de parceiros desenvolvedores e de contribuições de propriedade intelectual, isto é, uma grande quantidade de algoritmos compatíveis disponíveis para reuso.

3.2 TMS320 DSP Algorithm Standard [3][13]

Os Processadores Digitais de Sinais (também denominados *DSPs*) são muitas vezes programados como os microprocessadores 'tradicionais', onde predomina uma mistura de linguagens (C e Assembly) por razões como: performance, acesso direto a periféricos, etc. Na maioria das vezes é usado um pequeno ou mesmo nenhum sistema operacional. Assim como para os tradicionais microprocessadores, existe um uso muito pequeno, comercialmente (*comercial-off-the-shelf* - COTS), de componentes de *software* para *DSPs*.

Entretanto, diferentemente dos microprocessadores de propósito geral os *DSPs* são projetados para executar algoritmos sofisticados de processamento de sinais. Por exemplo, podem ser utilizados para reconhecimento de fala em um automóvel barulhento viajando a 100 km/h.

Tais algoritmos muitas vezes são o resultado de muitos anos de pesquisa. No entanto, por causa da falta de padrões consistentes é quase impossível reusar estes algoritmos em mais de um sistema sem uma significante reengenharia. Como poucas empresas podem dispor de um time de doutores em *DSP* e o reuso de algoritmos é muito trabalhoso, o tempo de lançamento no mercado de um novo produto baseado em *DSP* é medido em meses e muitas vezes, dependendo da complexidade, em anos.

As regras do *XDAIS* devem ser seguidas para que um algoritmo obedeça ao *eXpressDSP*. As *guidelines*, entretanto, são expressamente recomendadas mas não exigidas para que o algoritmo seja obediente ao padrão.

3.2.1 Exigências do padrão

A seguir são apresentadas as exigências do *TMS320 DSP Algorithm Standard*. Estas são utilizadas durante todo o documento para motivar escolhas de projeto e também ajudam a esclarecer a intenção de muitas regras e *guidelines*. As exigências do padrão são:

- Algoritmos de vários 'distribuidores' podem ser integrados em um sistema único. Um enorme número de algoritmos de processamento de sinais são necessários pelos mercados atuais (modems, reconhecedores de fala, cancelamento de eco, etc.). É impossível para um desenvolvedor de produtos que quer participar destes mercados, que possuem um grande conjunto de algoritmos, obter todos eles de uma única fonte. Por outro lado, a integração de algoritmos, de vários distribuidores é, muitas vezes impossível devido à incompatibilidades entre as várias implementações.

- Os algoritmos são livres de contexto, isto é, o mesmo algoritmo pode ser efi cientemente utilizado em, virtualmente, qualquer aplicação ou *framework*. Existem muitas empresas que utilizam diferentes *frameworks* em um mesmo mercado (por exemplo, o de telefonia) e, da mesma maneira cada distribuidor de componentes ou de algoritmos possui vários clientes. Para alcançar a meta do reuso de código, um mesmo algoritmo deve ser utilizável nos mais diversos *frameworks*.

- Os algoritmos podem ser utilizados em ambientes de execução puramente estáticos bem como dinâmicos. A fragmentação do mercado em vários *frameworks* distintos têm uma fundamentação técnica legítima, já que cada um otimiza a performance conforme a intenção de cada classe de sistema. Por exemplo, sistemas telefônicos clientes são projetados como um sistema de canal único com memória limitada, consumo limitado e com *DSPs* de baixo custo, um ambiente totalmente estático. Já em sistemas telefônicos servidores é possível utilizar um único *DSP*, manuseando múltiplos canais, reduzindo assim o custo por canal, mas para isso eles devem suportar um ambiente dinâmico. Em ambos os sistemas foram utilizados os mesmos algoritmos.

- Os algoritmos podem ser distribuídos em forma binária. É importante que os algoritmos sejam entregues em forma binária, isto não somente protege a propriedade intelectual do distribuidor do algoritmo como também melhora a reusabilidade. Se os fontes do algoritmo fossem necessários, todos os clientes teriam que recompilá-lo, além de que o controle de versão teria de ser feito pelo próprio cliente já que seria, por parte do fornecedor do algoritmo, inviável controlar os algoritmos de cada cliente.

- A integração dos algoritmos não exige recompilação da aplicação do cliente, embora, seja necessário reconfiguração e religação (*relinking*).

3.2.2 Objetivos do padrão

Embora nem sempre seja possível atingir esses objetivos perfeitamente, eles representam as principais preocupações que devem ser consideradas na definição dos elementos exigidos anteriormente.

- Facilidade para aderir ao padrão - Embora a TI desfrute de liderança no mercado de DSPs não controla diretamente os algoritmos de base. Isto é especialmente verdadeiro para uma família de *DSPs* (C54xx, processamento de imagem) onde existem algoritmos que são tecnologicamente importantes mas que não seguem nenhum padrão (muitas vezes são validações, implementadas, de pesquisas científicas que não têm qualquer preocupação em preparar o algoritmo para uma plataforma específica).

- Possibilidade de verificar conformidade ao padrão - Embora todos possam concordar com uma *guideline* que estabelece que todo o algoritmo deve ser de alta qualidade, este tipo característica não pode ser medida ou verificada. Esta impossibilidade de verificação ou mensuração permite aos integradores de sistemas reivindicar que todos os seus algoritmos são de alta qualidade e por isso não terão que pontuar (colocar um valor) esta *guideline*. Assim, é importante que cada uma seja mensurável ou, com alguma sensatez, verificável.

- Permitir aos integradores de sistemas uma fácil migração entre *DSPs* da Texas Instruments - Apesar deste padrão definir uma API (interface) para os algoritmos, de forma independente do *DSP*, ele não busca resolver o problema de migrações entre *DSPs*. Por exemplo, ele não exige que os mesmos algoritmos sejam fornecidos para duas plataformas distintas, ou seja, não especifica um formato de arquivo binário de forma a permitir que um binário único seja utilizado em dois projetos distintos. Também não fornece ferramentas para traduzir o código de uma arquitetura para outra ou exige o uso de uma linguagem independente da arquitetura (como o C) na implementação de algoritmos.

- Permitir ferramentas para simplificar tarefas do integrador do sistema, incluindo configuração, modelagem da performance, conformação ao padrão e depuração de erros - Onde possível, este padrão tenta antecipar as necessidades do integrador de sistemas e fornecer regras para o desenvolvimento de algoritmos que permitam a criação de ferramentas para assistir na integração destes algoritmos. Por exemplo, regras para convencionar a nomeação de algoritmos possibilitam ferramentas que, automaticamente, resolvam conflitos de nomes e selecionam implementações alternativas como apropriadas.

- Ficar sujeito a pouco ou nenhum *overhead* - Segundo Maurice Wikes [13], "Não há problemas na programação de computadores que não possam ser resolvidos pela adição de um nível de indireção". Os *frameworks* são exemplos de como a indireção é utilizada para resolver problemas arquiteturais de *software* para *DSPs*, a independência de dispositivo é realizada pela adição de um nível de indireção entre os algoritmos e os periféricos físicos, sendo o intercâmbio entre eles realizado pelo uso de ponteiros. Por outro lado, Jim Gray [13] disse, "Não há problema de performance que não possa ser resolvido pela eliminação de um nível de indireção".

3.2.3 Omissões intencionais

Aqui são citados aqueles aspectos do *Algorithm Standard* que foram intencionalmente omitidos neste texto, não que sejam sem importância mas no escopo de interesse desta dissertação não serão abordados:

- Controle de versão;
- Licenciamento, criptografi a e proteção de propriedade intelectual;
- Instalação e verifi cação;
- Documentação.

3.2.4 Divisão em níveis

O *TMS320 DSP Algorithm Standard* defi ne regras e *guidelines* (linhas guia) em três dos quatro níveis ilustrados na Figura 3.2 e descritos abaixo:

- Nível 1 - Contém as regras e linhas gerais de programação que são aplicadas a todos os algoritmos e arquiteturas de DSPs não importando a área de aplicação. Quase todos os módulos de *software* desenvolvidos recentemente seguem, de acordo comum, estas regras e este nível apenas formaliza isso.
- Nível 2 - Estabelece regras e *guidelines* que permitem a todos os algoritmos trabalharem harmoniosamente em um sistemas simples. Convenções são estabelecidas para o uso da memória de dados e nomes para identifi cadores externos pelos algoritmos, por exemplo. Adicionalmente, regras simples de como os algoritmos são 'empacotados' são também especifi cadas.
- Nível 3 - Contém as *guidelines* para famílias específi cas de DSPs. Atualmente não existe um acordo quanto ao uso de algoritmos com respeito ao uso dos recursos do processador. Estas

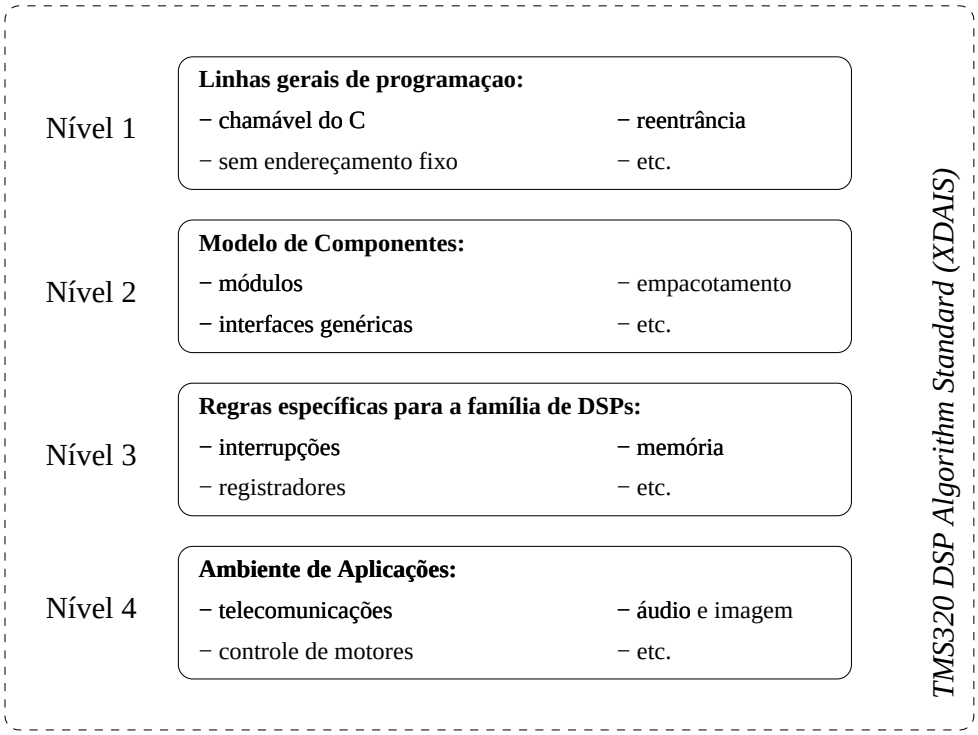

Figura 3.2: Elementos do *Algorithm Standard* [13]

guidelines irão prover uma direção sobre o que fazer ou não nas várias arquiteturas. Existe sempre a possibilidade de ocorrerem desvios destas *guidelines* mas, então, a especificação do algoritmo deve ser explícita, chamando atenção para os desvios com a documentação relevante.

- Nível 4 - Mostra os diversos mercados em que podem atuar os DSPs. Devido à natureza inerentemente diferente de cada um destes mercados, as interfaces são definidas como que para grupos de algoritmos, formalizando *frameworks* diversificados que têm como suporte o *XDAIS*.

3.2.5 Arquitetura do sistema

As várias arquiteturas de sistemas para *DSPs* podem ser particionadas conforme a Figura 3.3 e descrito como segue.

Os algoritmos são ‘puramente’ transdutores de dados, isto é, eles simplesmente ‘consomem’ os dados de entrada e ‘produzem’ dados de saída. O núcleo de suporte tempo real inclui funções de cópia de memória e funções que habilitam e desabilitam interrupções. O *framework* é a ‘cola’ que integra os algoritmos com as fontes de dados utilizando o núcleo de suporte tempo real, criando um subsistema completo para o *DSP*. Muitas vezes os *frameworks* dos *DSPs* interagem com os periféricos (incluindo outros processadores) e com freqüência definem as interfaces de E/S para os componentes de *software*.

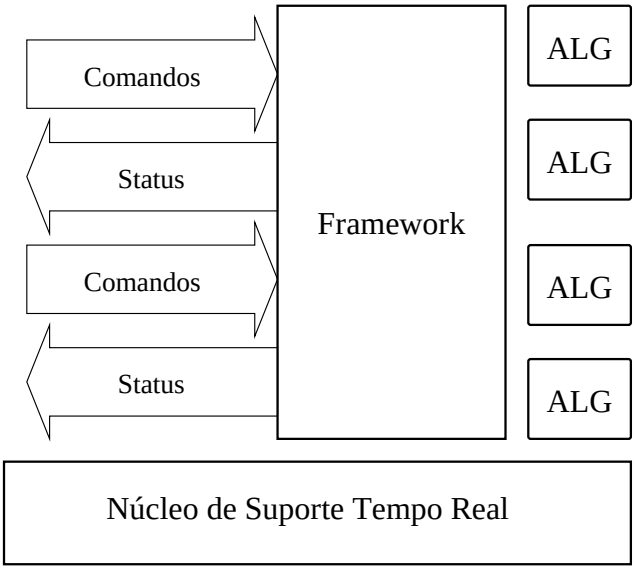

Figura 3.3: Arquitetura do *Software* para *DSPs* [13]

Infelizmente, por razões de performance vários sistemas para *DSPs* não apresentam uma linha clara dividindo os algoritmos e o nível de sistema, tornando assim, o reuso de algoritmos em mais de um sistema uma tarefa muito difícil e muitas vezes impossível. O *TMS320 DSP Algorithm Standard* é direcionado para definir esta linha de maneira a não sacrificar a performance e realçar, significativamente, a reusabilidade do algoritmo.

3.2.5.1 *Frameworks* e Algoritmos

Os *frameworks* com freqüência definem um subsistema independente de E/S do dispositivo e especifica como os algoritmos essenciais interagem com este subsistema. Por exemplo, é o algoritmo que chama a funções para requisitar os dados ou é o *framework* que chama o algoritmo com os dados já previamente alocados (*buffers*). Eles também definem o grau de modularidade dentro da aplicação, isto é, quais componentes podem ser substituídos, adicionados, removidos e quando isso irá acontecer (tempo de compilação, ligação ou execução).

As aplicações, na sua maioria, possuem em algum nível, componentes de *software*, compostos por algoritmos que compartilham de vários atributos comuns, tais como:

- São chamáveis do C;
- São reentrantes;
- São independentes de qualquer periférico de E/S específico;

- São caracterizados por suas exigências de memória e de desempenho em *MIPS* (*Milion Instructions Per Second*).

Em aproximadamente metade dos *frameworks* os algoritmos são também exigidos para simplesmente passar dados para um código específi co. Outros assumem que eles seriam ativados para adquirir um dado, por uma chamada específi ca de funções de E/S. Em todos os casos os algoritmos são projetados para serem independentes dos periféricos de E/S no sistema.

Em um esforço de minimizar as dependências, este padrão exige que algoritmos que processam dados recebam estes por parâmetro. Isto tem o objetivo de converter um algoritmo 'ativo' (que antes acessava diretamente o E/S) para um 'passivo' (que simplesmente aceita dados sob a forma de parâmetros), incorrendo em pouca ou nenhuma perda de performance.

Dadas as similaridades entre diversos *frameworks* parece possível padronizar o nível do algoritmo. Além disso, existe um benefício real para os distribuidores e integradores de sistemas: o tempo de integração é reduzido e facilita a comparação de preços entre o 'melhor' algoritmo e muitos outros disponíveis.

É importante compreender que cada implementação particular representa uma escolha dentro de um conjunto complexo entre tamanho de código, tamanho de dados, *MIPS* e qualidade. Além disso, dependendo do sistema projetado o integrador do sistema pode preferir um algoritmo com baixa qualidade e 'pequeno' a um com alta qualidade e muito 'maior' (por exemplo, um brinquedo eletrônico versus um sistema de correio de voz). Assim, faz sentido múltiplas implementações de um mesmo algoritmo, ou seja, não existe uma melhor e única implementação para este algoritmo.

Infelizmente, o integrador do sistema precisa muitas vezes escolher todos os algoritmos de um único distribuidor para assegurar a compatibilidade entre eles e minimizar o *overhead* de gerenciar APIs diferentes. Além disso, um único distribuidor não possui, muitas vezes, todos os algoritmos necessários para uma aplicação e o integrador do sistema se depara como tendo que escolher um distribuidor que tenha a 'maioria' dos algoritmos exigidos e negocia com ele a implementação dos restantes.

Possibilitar ao integrador do sistema conectar ou substituir um algoritmo por outro reduz o tempo para o produto chegar ao mercado e também torna o sistema mais flexível já que ele pode escolher algoritmos de múltiplos distribuidores.

3.2.5.2 Núcleo de suporte tempo real

Para possibilitar aos algoritmos satisfazerem as exigências mínimas de reentrância, independência de periféricos de E/S e depurabilidade os algoritmos devem contar com um conjunto de serviços que

estão sempre presentes. Já que muitos algoritmos ainda são produzidos em linguagem *Assembly* muitos dos serviços providos pelo núcleo devem ser acessíveis e apropriados para esta linguagem.

O núcleo de suporte tempo real inclui um subconjunto de funções que suportam modificações atômicas de registradores de controle/status. Ele também inclui um subconjunto de funções da linguagem C ANSI com suporte tempo real (exemplo: *memcpy*, *strcpy*, etc.).

3.3 Kernel DSP/BIOS [28][26]

Tradicionalmente as aplicações para *DSPs* eram muito simples, tipicamente utilizando um único programa para executar o processamento necessário. Ao longo do tempo, com o aumento da complexidade, as aplicações começaram a ter a necessidade de processamento concorrente e atualmente têm se tornado crítico que o *DSP* faça diversas tarefas simultaneamente. Além disso, as aplicações também evoluem (melhoramentos nos algoritmos de controle, implementação de mais funções como comunicação e outras) o que exige suporte para adicionar ou modificar suas características. Construir estas modernas aplicações usando o tradicional paradigma do laço único é, não só desafiador, como também torna muito difícil a manutenção.

Atualmente os desenvolvedores de sistemas gastam mais tempo projetando *software* que projetando *hardware*. Isto acontece devido ao fato que muitas funcionalidades, complexas em *hardware*, vêm sendo embutidas nos circuitos integrados. Os desenvolvedores de sistemas simplesmente precisam escolher o *chip* certo para trabalho.

Para implementar funções de processamento de sinais os projetistas podem, ou utilizar circuitos integrados de *hardware* dedicado, ou *DSPs* (processadores programáveis para esse fim). As vantagens de utilizar *DSPs* são:

- Escalabilidade - os projetistas primeiro decidem quanto de poder de processamento (*MIPS - Million Instructions Per Second*), memória e outros recursos são necessários para determinado trabalho e então selecionam o *DSP* (ou *DSPs*) que possuem os recursos necessários.
- Atualização - Depois de fixado o projeto de *hardware* qualquer mudança nas funções de processamento de sinais são facilmente feitas em *software* e o novo programa é descarregado no *DSP* em segundos.

Entretanto, ao contrário dos *hardware* dedicados, que implementam funções para o processamento de sinais, os *DSPs* precisam gerenciar os recursos que podem ser compartilhados entre as diferentes funções, entre eles estão:

- CPU;

- Memória;
- Periféricos.

O *software* exigido em um típico sistema embutido usando *DSP* engloba dois componentes: o *software* da aplicação e o *software* de sistema como pode ser visto na Figura 3.4

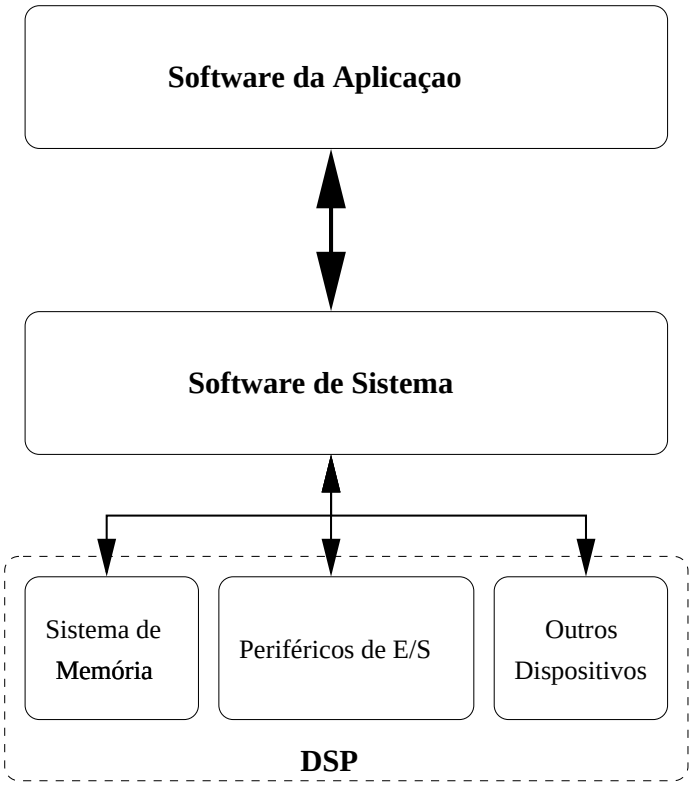

Figura 3.4: Típico sistema embutido em *DSPs* [26].

O *software* de sistema é responsável por gerenciar os recursos do sistema para a aplicação. Como recursos do sistema pode-se entender todos os dispositivos de *hardware* da plataforma alvo, neste caso o *DSP*. O *software* de sistema é quem provê infra-estrutura para a aplicação. Todas as aplicações com *DSPs* requerem algum *software* de sistema para gerenciar os seus recursos, ele gerencia o acesso entre o *software* da aplicação e os recursos do sistema (*hardware* físico) provendo uma camada de abstração entre eles.

O kernel DSP/BIOS é um pequeno *firmware*, proposto pela Texas Instruments, que executa em um processador digital de sinais (*DSP*) e que provê componentes de *software* de forma a permitir, aos desenvolvedores de aplicações, a capacidade de:

- Monitoração e controle em tempo real - monitorar e controlar a execução e as variáveis do programa em tempo de execução;

- Escalonamento tempo real - escalonamento e comunicação em sistemas *multi-thread* em tempo de execução.

Em sistemas simples o *software* de sistema consiste de uma inicialização básica de *hardware*, funções de acesso a periféricos e rotinas de tratamento de interrupção (*ISRs - Interrupt Service Routines*). Já os sistemas mais complexos exigem um escalonamento para garantir a correta operação das diversas funções (algoritmo de controle, comunicação via rede, etc.) dependendo da prioridade de cada um. Além disso, as aplicações muitas vezes exigem acesso concorrente aos recursos de *hardware* (memória, E/S, etc.). Gerenciar estes recursos é precisamente um dos benefícios de se utilizar o kernel DSP/BIOS, entre outros tem-se:

- Gerenciar os *MIPS* do *DSP* efi cientemente através da utilização de *threads* múltiplas;
- Utilizar interfaces padrões para interrupções de E/S e de *hardware*;
- Defi nir e confi gurar recursos do sistema efi cientemente;
- Adicionar estruturação para a aplicação, organizando isto através de um conjunto de *threads* inter-relacionadas;
- Tornar fácil a migração para novos processadores, já que grande parte do *hardware* é abstrato para a aplicação.

Em resumo, o DSP/BIOS é um produto da *Texas Instruments Inc.* que compreende um núcleo de tempo real projetado para aplicações que requerem escalonamento tempo real e sincronização, comunicação e instrumentação. Fornece um ambiente *multi-thread* preemptivo, abstração de *hardware* e ferramentas de confi guração, provendo uma coleção de serviços que os desenvolvedores usam para gerenciar os recursos em nível de sistema e construir a infra-estrutura para as aplicações com *DSPs*. Esses recursos são ajustados e otimizados de acordo com necessidades com tamanho e/ou performance, o DSP/BIOS, atualmente, está disponível para os *DSPs* das famílias TMS320C5000 e TMS320C6000.

3.3.1 Características principais do DSP/BIOS

De forma a entender o comportamento do kernel DSP/BIOS procurou-se abranger aqui suas principais características.

3.3.1.1 Configuração gráfica

Como muitos sistemas operacionais de tempo real, o DSP/BIOS possibilita a uma aplicação a criação dinâmica de objetos do sistemas operacional, como *threads* ou semáforos, a qualquer momento

durante a execução do programa. Em adição a isto o DSP/BIOS provê uma ferramenta gráfica de configuração que permite um acesso simples a configuração necessária de uma aplicação. Este processo de configuração estático reduz a memória final a ser utilizada pelo *DSP*, otimizando e eliminando funções do núcleo não utilizadas pela aplicação.

3.3.1.2 Módulos e Serviços

Para incrementar o suporte para análises e configuração de periféricos o DSP/BIOS inclui em seu núcleo os seguintes serviços:

- Interrupções de *hardware* - interface entre as interrupções de *hardware* e o núcleo do DSP/-BIOS;
- Interrupções de *software* - *threads* 'leves' que usam a pilha de programa e não sofrem preempção;
- *Tasks* - *threads* independentes da execução principal (*background*) que podem ceder o processador;
- Funções periódicas - *threads* 'leves' disparadas em tempos determinados;
- *Mailboxes* - sincronizam a troca de dados entre *threads*;
- Semáforos
- Filas - listas ligadas atômicas;
- *Clock* - interface dos *timers* de *hardware*;
- *Streams* para comunicação entre *threads* e dispositivos de E/S;
- Gerenciador de memória - baixo *overhead* na alocação dinâmica de memória.

Para prover a resposta rápida exigida por uma aplicação com *DSP*, o DSP/BIOS aumenta o tradicional modelo de tarefas com mecanismos adicionais. As interrupções de *software* são *threads* 'leves' que compartilham uma pilha comum. Isto resulta em um baixo *overhead* da memória e trocas de contexto mais rápidas já que não é necessário salvar e restaurar a pilha de tarefas. Funções periódicas são disparadas como *threads* de alta prioridade que podem facilmente processar os dados amostrados em intervalos fixos de tempo, simplificando o projeto de sistemas que possuem várias taxas de amostragem (*multirate*). Para facilitar o projeto de aplicações sofisticadas, o DSP/BIOS provê serviços de comunicação entre *threads*, incluindo semáforos, *mailboxes* e filas.

O *DSP/BIOS* provê módulos que permitem aos projetistas de sistemas realizar tarefas essenciais e em tempo de execução tais como: monitoração e controle, e escalonamento e comunicação. Tais módulos podem ser visualizados na Tabela 3.1 e descritos a seguir.

Módulos para confi guração do sistema	
GBL	*Global setting manager*
MEM	*Memory manager*
Módulos para controle e monitoração tempo real	
LOG	*Message log manager*
STS	*Statistic accumulator manager*
TRC	*Trace manager*
Módulos para escalonamento tempo real	
HWI	*Hardware interrupt manager*
SWI	*Software interrupt manager*
IDL	*Idle function and processing loop manager*
CLK	*System clock manager*
PRD	*Periodic function manager*
Módulos para comunicação tempo real	
PIP	*Data pipe manager*
HST	*Host input/output manager*
RTDX	*Real-time data exchange manager*

Tabela 3.1: Módulos principais do Kernel DSP/BIOS [28]

- Módulos para confi guração do sistema - defi ne o *hardware* e o ambiente do sistema;
- Módulos para controle e monitoração tempo real - provê um meio de enviar informações para o hospedeiro (*host* - o PC, por exemplo) enquanto o programa executa no *DSP*;
- Módulos para escalonamento tempo real - tem por função manusear uma *thread* particular em um ambiente tempo real. Estas rotinas podem ser algoritmos de controle implementados pelo desenvolvedor do sistema, por exemplo;
- Módulos para comunicação tempo real - responsável por gerenciar os canais de comunicação entre *threads* e entre o DSP e o hospedeiro (PC, por exemplo).

Utilizando os módulos do DSP/BIOS os projetistas de sistemas podem desenvolver algoritmos de controle usando diagramas de blocos, onde cada bloco representa uma *thread*, similar ao que é encontrado no projeto de *hardware* utilizando circuitos integrados dedicados como blocos.

3.3.1.3 Gerenciamento de interrupções

O DSP/BIOS provê duas opções para o manuseio de interrupções, um ‘despachante’ de interrupções e macros. As duas implementações tomam o cuidado de minimizar a tempo em que as interrupções premanecem desabilitadas para otimizar a latência da interrupção. O ‘despachante’ permite que as *ISRs* sejam escritas em C fazendo operações como salvamento e restauração do contexto

e desabilitando as interrupções do escalonador temporariamente, o que permite que as *ISRs* possam interagir corretamente com o núcleo. Esta abordagem reduz o tamanho do código eliminando a necessidade de inclusão de rotinas específicas dentro das *ISRs*. Para reduzir os requisitos de memória, todas as *ISRs* utilizam uma pilha comum. A entrada e a saída de macros podem ser adicionadas à uma *ISR* para executar as mesmas operações de um 'despachante'. Embora as funções macro devam ser duplicadas em cada *ISR*, elas permitem ao desenvolvedor otimizar as operações de salvamento e restauração de registradores para uma *ISR* específica, minimizando assim o tempo de resposta. O DSP/BIOS habilita o uso das duas abordagens que podem ser utilizadas na mesma aplicação para permitir um ótimo balanceamento entre performance e utilização de memória.

3.3.1.4 Abstração de *hardware*

Em adição aos serviços de escalonamento e comunicação entre *threads*, o DSP/BIOS também provê um gerenciador de *timers* em tempo real, um gerenciador de memória e dispositivos de E/S independentes. O gerenciador de memória do DSP/BIOS provê um reentrante e dinâmico gerenciamento de memória que abstrai facilmente as configurações fragmentadas de memória típicas das aplicações com *DSPs* que utilizam segmentos da memória lógica. Também possui um modelo de dispositivos de E/S independentes que é otimizado por um processamento eficiente de *streams* de dados. A abstração mantém longe as dependências dos dispositivos periféricos e as configurações de memória, o DSP/BIOS torna mais simples migrar as aplicações para novos *DSPs*.

3.4 *Framework* de Referência [2]

Com a introdução das ferramentas e metodologias vistas (DSP/BIOS, XDAIS e Code Composer Studio), a Texas Instruments resolveu direcioná-las para o que chamou de '*Reference Framework*'.

Os *Reference Frameworks* provêm um ponto de partida para aplicações que utilizam o *DSP/BIOS* e o *TMS320 Algorithm Standard*. Os desenvolvedores primeiramente selecionam o *Reference Framework* que melhor se aproxima do seu sistema e de suas necessidades futuras e então adaptam ele populando-o com algoritmos obedientes ao padrão. Os elementos comuns como *drivers* de dispositivos, gerenciador de memória e outros já estão pré-configurados nestes *frameworks*. Os desenvolvedores podem focar unicamente nas necessidades da aplicação aumentando consideravelmente a produtividade.

3.4.1 Características das arquiteturas de aplicações com *DSP*

Algumas características são necessárias para se determinar uma arquitetura comum, apropriada para a maioria dos sistemas de um determinado *framework*. É importante notar que neste ponto

as decisões arquiteturais são independentes dos detalhes das aplicações finais. Sendo assim é bom considerar as seguintes características:

- Quantos algoritmos são utilizados no sistema?
- O sistema exige criação dinâmica de objetos e alocação de recursos ou uma configuração estática é suficiente?
- O sistema irá executar em uma taxa de freqüência ou em várias taxas?
- Existem restrições quanto a quantidade de memória?
- O sistema necessita de controle externo?

As respostas a essas perguntas esclarecem como o sistema será construído, compondo um *framework* de base, isto é, um *Reference Framework*. Entretanto, não se consegue saber nada sobre a aplicação final ou quais algoritmos serão utilizados no sistema.

3.4.2 Características comuns dos *Frameworks*

Antes de discutir os detalhes de cada *Reference Framework* é importante considerar as características compartilhadas por todos os *frameworks*.

- Código 100% em C - Os códigos do *Reference Framework* são 100% escritos em linguagem C, se tornando assim fáceis de modificar.
- Soluções completas - Cada *Reference Framework* é provido como uma solução completa. Parte da estratégia é mostrar como é fácil que algoritmos simples possam ser substituídos por componentes mais complexos. Isto se torna possível através do uso do *XDAIS*.
- Fácil adaptação - Os desenvolvedores conseguem passar, rapidamente, de exemplos genéricos para aplicações específicas sem necessariamente terem de entender os detalhes de 'baixo nível' do sistema.
- Documentação - Detalhamentos em termos de velocidade de processamento e exigências de memória.

3.4.3 Definindo os níveis dos *Reference Frameworks*

Categorizando os sistemas de acordo com as respostas das perguntas da seção anterior permitiu dividir os *Reference Frameworks* (RF) em níveis, conforme pode ser visto na Tabela 3.2, e descritos

a seguir. A diferença fundamental entre cada RF são as quantidades de módulos do DSP/BIOS presentes, o que diminui as funcionalidades e o suporte do sistema mas também mantém o código total bastante otimizado.

Parâmetro de projeto	RF1	RF3	RF5
Confi guração estática	√	√	√
Criação dinâmica de objetos	-	-	-
Gerenciamento de memória estática	√	√	√
Alocação dinâmica de memória	-	√	√
Número de canais	1 a 3	1 a 10	1 a 100
Número de algoritmos	1 a 3	1 a 10	1 a 100
Memória absoluta mínima	√	-	-
Única taxa de operação	√	√	√
Múltiplas taxas de operação	-	√	√
Threads bloqueantes	-	-	√
Funcionalidade de controle externo	-	√	√

Tabela 3.2: *Reference Frameworks* caracterizados por nível [2].

- RF1 (*Framework* Compacto) - Este *framework* é projetado com o mínimo necessário. Utiliza confi guração estática e não suporta criação dinâmica de objetos. O gerenciamento de memória é completamente estático e não suporta gerenciamento dinâmico. Não há preempção nem bloqueio de *threads* também não provê módulos do DSP/BIOS para controle ou comunicação.

- RF3 (*Framework* Flexível) - Difere, signifi cativamente, do nível anterior. O mínimo necessário é substituído pela flexibilidade. Suporta somente criação estática de objetos mas a criação dinâmica pode ser adicionada. Os *buffers* de dados podem ser confi gurados e gerenciados em tempo de execução. Suporta também múltiplas taxas de operação, isto quer dizer que diferentes algoritmos podem executar em diferentes freqüências (um a 10ms outro a 20ms, por exemplo). Provê também uma *thread* adicional que permite controle externo do *DSP*.

- RF5 - (*Framework* Estendido) - Este *framework* é concebido para projetistas que procuram uma flexibilidade estendida sem se preocupar com restrições do processador. Suporta criação estática e dinâmica de objetos e também gerenciamento de memória estático e dinâmico. Suporta uma grande quantidade de algoritmos e permite múltiplas taxas de operação. Normalmente este nível requer mais de 70% dos módulos do *DSP/BIOS* para que o sistema suporte todas as funcionalidades requeridas por este RF.

3.4.4 Elementos de um *Reference Framework*

A Figura 3.5 ilustra os elementos que compõe o *Reference Framework*, tendo como alvo o *DSP*.

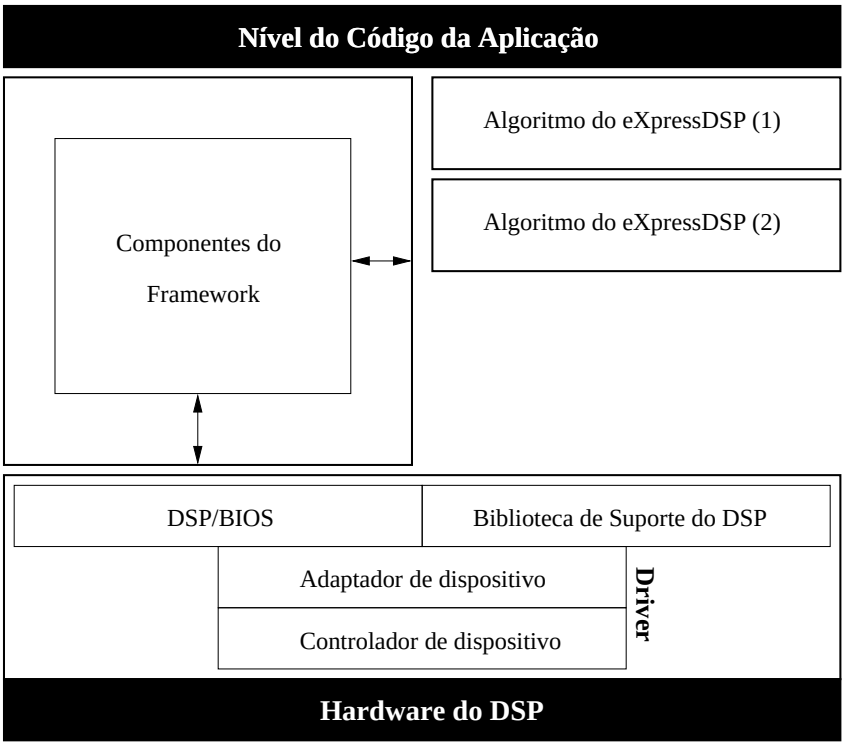

Figura 3.5: Elementos de um *Reference Framework* [2].

Abaixo segue uma breve descrição dos elementos haja visto que os mais importantes já foram abordados neste capítulo.

- Controlador e Adaptador de dispositivo - Os *drivers* de dispositivos utilizados no *Reference Framework* são baseados no modelo padrão de *driver*, o qual provê adaptadores de dispositivo e especifi ca uma interface padrão do controlador de dispositivo. Considerando o *hardware* alvo é unicamente necessário customizar o controlador de dispositivo enquanto que o adaptador necessita de pouca ou nenhuma modifi cação. O ponto chave deste modelo é a separação entre a aplicação e o *dirver*.

- Biblioteca de suporte do *chip* - Biblioteca utilizada pelo controlador de dispositivo para suportar os periféricos de *hardware*.

- DSP/BIOS - Um *software* de sistema extensível que provê infra-estrutura para a aplicação cliente. A quantidade de módulos utilizados num determinado *Reference Framework* vai depender do nível deste, isto é, a quantidade de funções necessárias para prover uma infra-estrutura adequada para o código cliente.

- Componentes do *framework* - Estes elementos são concebidos para prover todo o gerenciamento de recursos de um determinado sistema. Dependendo do *Reference Framework* escolhido os componentes mudam, ou seja, escolhendo um *RF* mais simples os componentes ten-

dem a otimizar a memória utilizada e um *RF* mais complexo tende a possuir componentes que privilegiam a flexibilidade, por exemplo.

- Algoritmos obedientes ao eXpressDSP - Cada algoritmo deve seguir, como já foi explicado, as regras e linhas-guia do *TMS320 DSP Algorithm Standard* (*XDAIS*).
- Nível do código da aplicação - A última coisa a ser modificada. Neste nível está aplicação específica baseada no conhecimento do desenvolvedor permitindo a verdadeira diferenciação do produto.

3.5 Projeto de Software utilizando Diagramas de Blocos [8]

Dentro do espaço dos Sistemas de Controle Digitais (*Digital Control Systems - DCS*) o *Algorithm Standard* tem sido utilizado para necessidades específicas de áreas de aplicação como Controle Digital de Motores (*Digital Motor Control - DMC*), Automação Industrial (AI), Fornecedores Ininterruptos de Energia (*Uninterruptible Power Supplies - UPS*) e muitas outras áreas de aplicação do controle. A familiar representação por diagramas de blocos de sistemas digitais de controle é utilizada para se visualizar a interconexão entre blocos funcionais (módulos), percebendo-se intuitivamente as entradas e saídas.

Uma vez que um grande conjunto de módulos está disponível, seguindo uma metodologia que comprova e define a interconexão entre eles, os sistemas podem ser então implementados facilmente e rapidamente. Com isso vários tipos de topologias ou configurações de sistemas podem ser explorados. Como uma necessidade fundamental, as conexões de um mesmo módulo permitem ao desenvolvedor possuir um número incremental de níveis de 'construção', desde um sistema inicial (3 a 4 módulos, para verificação de sinais vitais) até um sistema final com inúmeros módulos conectados.

É bem conhecido que engenheiros de controle preferem visualizar os sistemas e as estratégias de controle na forma de fluxos de sinais em diagramas de blocos. Isto faz com que sejam separadas as funções principais (PIDs, transformações, integrações, etc.) e mostre como cada uma se relaciona, através de conexões explícitas, com outras funções. Cada função deve ser auto-contida e seus limites e interface bem delineados. Um sistema para controle digital de motores é um bom exemplo disto e a Figura 3.6 mostra um típico diagrama de blocos orientado para esta estratégia de controle. Esta é uma representação usual e classicamente encontrada em muitos textos sobre DMC.

Entretanto, as limitações são evidentes quando se tenta transformar este diagrama em uma implementação de *software*. Não é usual mostrar variáveis de *software* como parâmetros de sinais, além disso, a fronteira entre *software* e *hardware* não é clara, por exemplo, quando o *software* controla um periférico do CI do *DSP* onde os pinos de saída do periférico controlam um *hardware* externo.

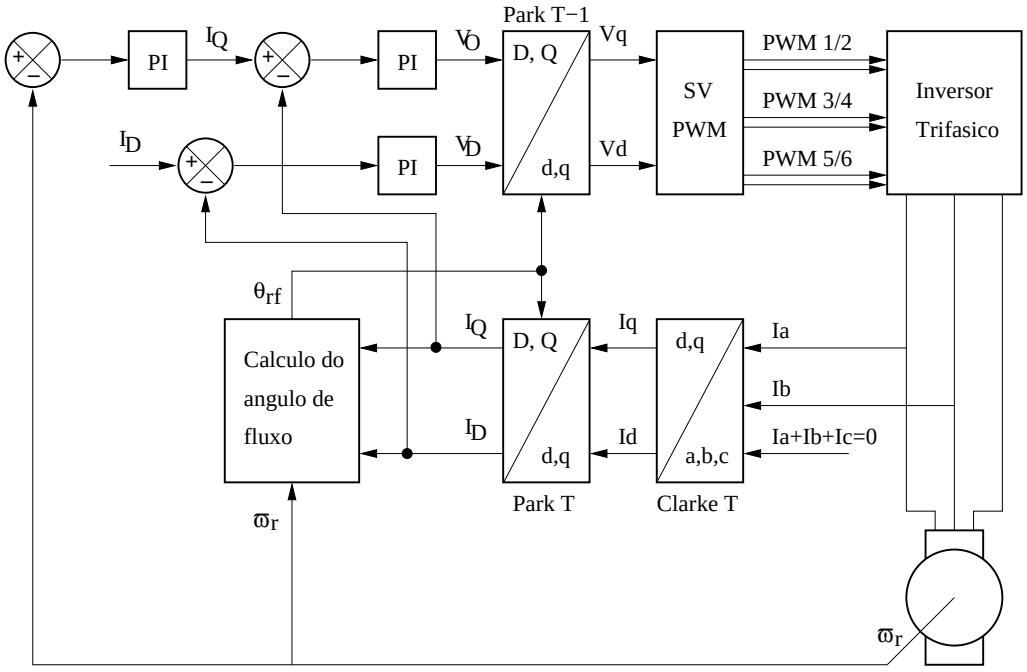

Figura 3.6: Clássico Sistema de Diagrama de Blocos [8]

Para adotar a estratégia modular e clarear as definições, o diagrama de blocos clássico ilustrado na Figura 3.6 deve ser rearranjado e redesenhado de maneira a revelar a grande quantidade de informação sobre o *software* que implementará o sistema. O novo 'estilo' de representação do sistema pode ser visto na Figura 3.7 e suas vantagens podem ser resumidas nos seguintes pontos:

- O diagrama de blocos do sistema tem um mapeamento 1-para-1 no sistema modular de *software*.
- Cada bloco (ou módulo) representa uma função de *software* auto contida.
- Os terminais de entrada e saída de cada módulo correspondem, exatamente, às variáveis globais passadas para uma função de *software*.
- Os módulos são classificados de forma a ilustrar as dependências de periféricos e *hardware*.
- As conexões entre os módulos mostra o fluxo de dados e as correspondentes variáveis de entrada e saída.
- Cada módulo é reusável e tem sua própria documentação explicando a utilização.

3.5.1 Classificação dos módulos de acordo com a dependência de periférico e *hardware*

Entender a exata dependência de um módulo de *software* é muito importante, pois se for necessário mudar a plataforma utilizada sabe-se exatamente quais módulos devem ser substituídos ou

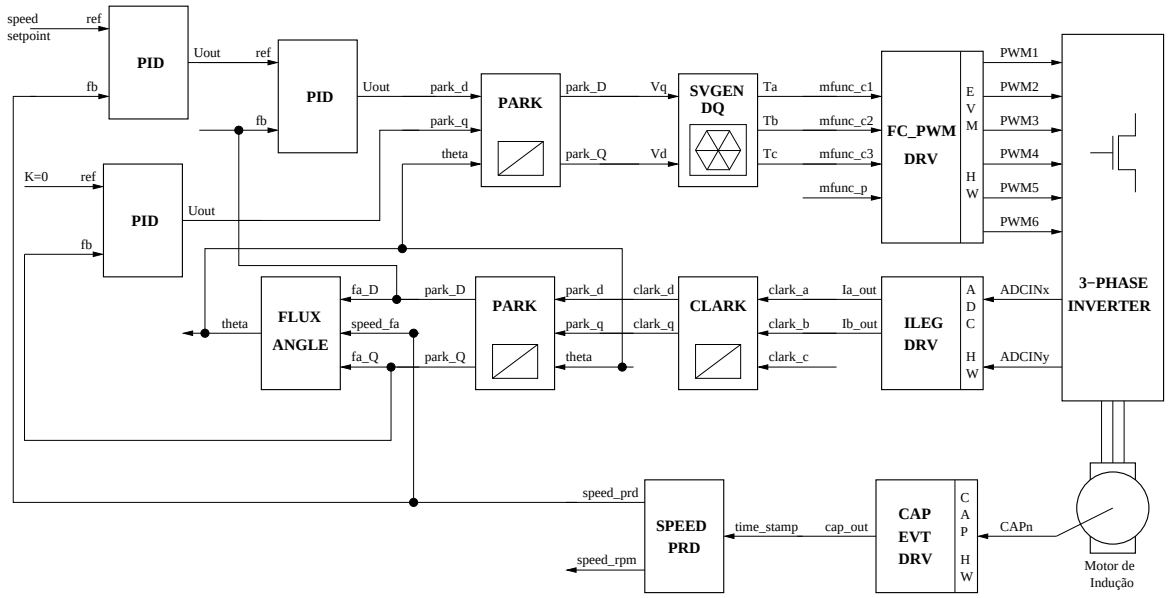

Figure 3.7: Diagrama de Blocos para representação como Módulos de *Software* [8]

modificados e quais podem permanecer inalterados. Os módulos que fazem parte da biblioteca de *software* dos DCS podem ser divididos como segue, de acordo com a utilidade ou a dependência ou não da aplicação e *hardware*, e são ilustrados nas figuras referenciadas:

- Módulos Utilitários e de Depuração - são usados durante os processos de desenvolvimento e depuração do *software*. Tipicamente, são removidos quando o *software* é finalizado, entretanto, também pode ser deixado para diagnóstico do sistema durante testes ou avaliações (Figura 3.8).
- Independente de *Hardware*/Independente da Aplicação - são tipicamente caracterizados por funções matemáticas, normalmente fixas e padronizadas e não requerem configuração, conhecimento dos periféricos ou da aplicação em si (Figura 3.9).
- Independente de *Hardware*/Dependente da Aplicação - estes módulos também não necessitam conhecer os periféricos do dispositivo mas, necessitam ser configurados pela aplicação principal. Exemplos destes módulos são os controladores PID, estimadores de velocidade, voltagem e modelos de corrente (Figura 3.10).
- *Drivers* (Dependente do *Hardware* e da Aplicação) - são a interface entre o *software* e um periférico específico do *hardware* do *DSP*. Estes módulos têm acesso direto aos registradores de controle e *status* do periférico e são dependentes de configuração pela aplicação (Figura 3.11).

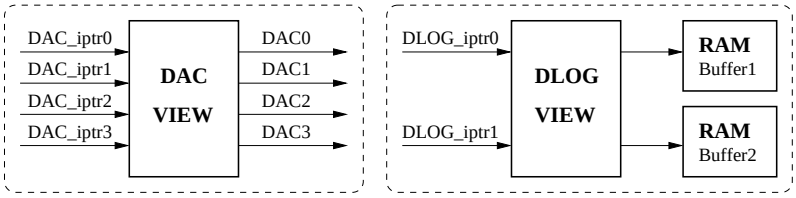

Figure 3.8: Exemplos de módulos Utilitários e de Depuração [8][11]

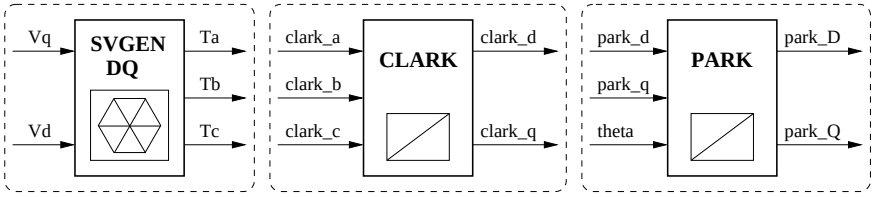

Figure 3.9: Exemplos de módulos independentes do *Hardware* e da Aplicação [8][11]

3.5.2 Exemplo de implementação baseada em blocos

Para melhor ilustrar os conceitos até aqui descritos será inserido um exemplo de um controlador desenvolvido no Departamento de Automação e Sistemas (DAS) para o controle em malha fechada de um motor. Este controlador foi desenvolvido como parte do trabalho de iniciação científi ca do acadêmico Leonardo Augusto Weiss, aluno do curso de Engenharia de Controle e Automação nesta Universidade.

Serão descritos os módulos utilizados e seu encapsulamento através do *Algorithm Standard* e por fi m como o sistema foi composto resultando no que pode ser visto na Figura 3.12, no esquemático de blocos, parte de *software*. É importante ressaltar que esta família de *DSPs* (família 2000) não possui kernel DSP/BIOS[2] tendo a aplicação que se ater à execução de todas as funções na ocorrência da *ISR*.

[2]O desenvolvimento de um micronúcleo que possibilite suporte à multiplas tarefas é objetivo desta dissertação.

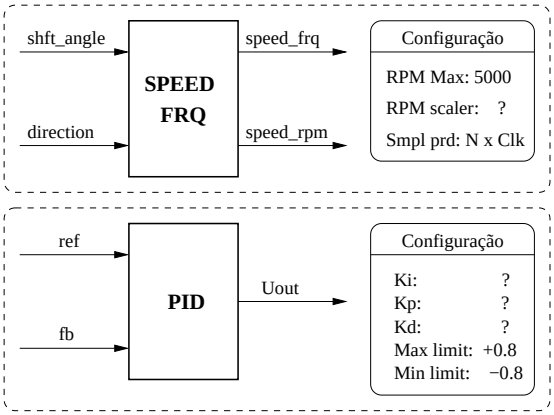

Figure 3.10: Exemplos de módulos independentes do *Hardware* e dependentes da Aplicação [8][11]

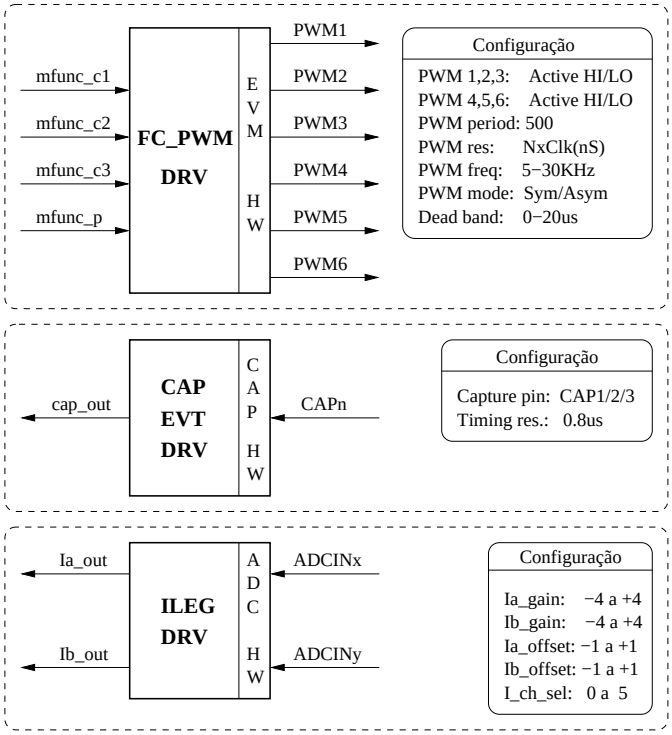

Figure 3.11: Exemplos de módulos dependentes do *Hardware* e da Aplicação - *Drivers* [8][11]

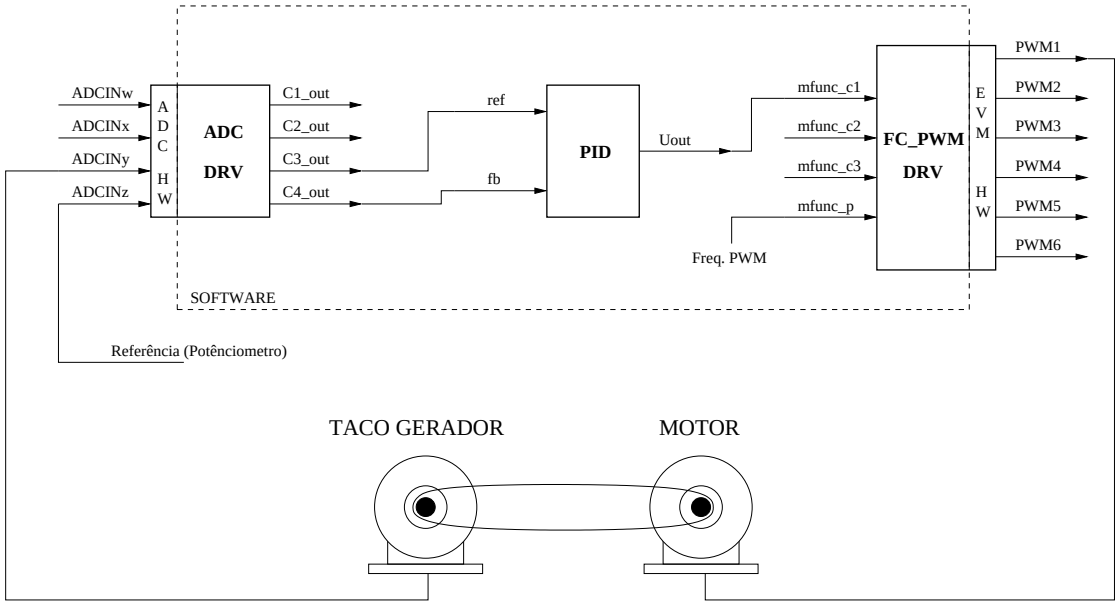

Figura 3.12: Esquemático do sistema construído

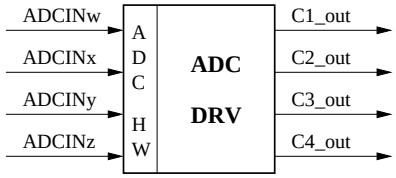

Figura 3.13: Módulo ADC[11]

Os módulos utilizados na aplicação do controlador, podem ser encontrados juntamente com dezenas de outros na biblioteca de controladores digitais de motores [11].

3.5.3 Módulo ADC (Figura 3.13)

O conversor A/D (ADC - *Analogic Digital Converter*) do DSP possui 16 canais de 10 bits de resolução e, na aplicação, é implementado pela biblioteca ADC04U_DRV. Este módulo entrega valores entre 0 e 32767 correspondendo a tensões de entrada de 0 e 3.3V, respectivamente. Para obter essa faixa de valores, o valor convertido pelo ADC é rotacionado 5 bits à esquerda. Toda a configuração do conversor é abstraída, e os valores convertidos passam a figurar como variáveis que podem ser facilmente manipuladas no código do programa.

O ADC possui uma interface (Algoritmo 3.1) onde estão definidas as variáveis necessárias para iniciar e configurar o módulo.

Algoritmo 3.1: Interface do módulo ADC04U_DRV[11]

```
typedef struct
{
        int c1_gain; /* Gain control for channel 1 */
        int c2_gain; /* Gain control for channel 2 */
        int c3_gain; /* Gain control for channel 3 */
        int c4_gain; /* Gain control for channel 4 */
        int c1_out;  /* Conversion result for channel 1 */
        int c2_out;  /* Conversion result for channel 2 */
        int c3_out;  /* Conversion result for channel 3 */
        int c4_out;  /* Conversion result for channel 4 */
        int a4_ch_sel; /* ADC channel select variable */
        int (*init)(); /* Initialization function pointer */
        int (*update)(); /* Update function pointer */
} ADCVALS;
```

Para a correta operação do módulo ADC primeiramente cria-se a estrutura ADCVALS e carrega-se seus valores padrões, ADC_DEFAULTS. Deve-se escolher que canais deverão ser utilizados, isto é feito atribuindo-se um valor a variável *a4_ch_sel*. No sistema precisou-se adquirir dois sinais: a referência e a medição da velocidade. O manual faz menção que os canais 1 e 2 não devem ser

utilizados por apresentarem baixa resolução, então utilizou-se os canais 3 e 4, ajustando o valor da variável *a4_ch_sel* em 0x3210, em hexadecimal.

Para inicializar o *hardware* do conversor ADC utiliza-se o método *init(&adc)*. Para atualizar os valores lidos pelo ADC usa-se o método adc.update(&adc) e os valores convertidos passam a figurar como as variáveis *c3_out*, referente ao canal 3 e *c4_out*, referente ao canal 4.

Todo o procedimento é executado uma única vez na função *main*, com exceção da atualização do ADC (*update(&adc)*) que é realizada a cada ocorrência da *ISR* como pode ser visto no Algoritmo 3.2.

Algoritmo 3.2: Exemplo de utilização do módulo ADC04U_DRV[11]

```
ADCVALS adc = ADC_DEFAULTS; /* Instanciation and inicialization */

void main(void) /* Main function */
{
        adc.a4_ch_sel = 0x3210; /* Channel selection */
        adc.c3_gain = 0x1FFF; /* Channel 3 Gain */
        adc.c4_gain = 0x1FFF; /* Channel 4 Gain */

        adc.init(&adc); /* Hardware initialization */
}

void interrupt periodic_interrupt_isr() /* ISR function */
{
        adc.update(&adc); /* Update ADC values */
        x = adc.c3_out; /* Read ADC Channel 3 value */
        y = adc.c4_out; /* Read ADC Channel 4 value */
}
```

3.5.4 Módulo PID (Figura 3.14)

Este módulo implementa um controlador digital PID (Proporcional Integral Derivativo). Também pode ser usado como um controlador PI ou PD. Nesta implementação, a equação diferencial é transformada em uma equação de diferenças por meio de aproximação inversa.

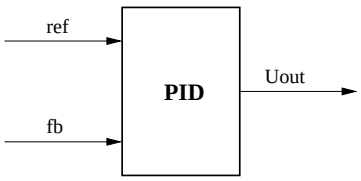

Figura 3.14: Módulo PID[11]

Assim como o ADC este módulo possui uma interface (Algoritmo 3.3) onde estão definidas as variáveis de inicialização e configuração do controlador PID.

Algoritmo 3.3: Interface do módulo PID_REG1[11]

```
typedef struct
{
        int pid_ref_reg1; /* Input: Reference input */
        int pid_fb_reg1;  /* Input: Feedback input */
        int Kp_reg1;      /* Parameter: Proportional gain */
        int Ki_high_reg1; /* Parameter: Integral gain */
        int Ki_low_reg1;  /* Parameter: Integral gain */
        int Kd_reg1;      /* Parameter: Derivative gain */
        int pid_out_max;  /* Parameter: Maximum PID output */
        int pid_out_min;  /* Parameter: Minimum PID output */
        int pid_e1_reg1;  /* History: Previous error at time = k-1 */
        int pid_e2_reg1;  /* History: Previous error at time = k-2 */
        int pid_out_reg1; /* Output: PID output */
        int (*calc)();    /* Pointer to calculation function */
} PIDREG1;
```

Para utilizá-lo cria-se a estrutura PIDREG1 e atribui-se a esta os valores padrões, dados por PIDREG1_DEFAULTS. As variáveis *Kp*, *Ki*, *Kd* correspondem aos ganhos do controlador PID contínuo e as *pid_out_max* e *pid_out_min* a saturação da ação de controle. O ganho integral possui dois valores, pois ocupa 32 bits, a parte menos signifi cativa desses bits correspondem ao *Ki_low_reg1* e a parte mais signifi cativa ao *Ki_high_reg1*. Outras três variáveis merecem atenção, a *pid_ref_reg1*, que corresponde ao valor de referência para o controlador, a *pid_fb_reg1*, que representa a realimentação do controlador e *pid_out_reg1*, que é a ação de controle. Passados os parâmetros necessários ao controlador, o método que inicia o cálculo da ação de controle é o *calc(&pid)*[3].

O Algoritmo 3.4 ilustra a utlização do módulo PID.

Algoritmo 3.4: Exemplo de utilização do módulo PID_REG1[11]

```
PIDREG1 pid = PIDREG1_DEFAULTS /* Instance the object */

void main(void) /* Main function */
{
        pid.pid_ref_reg1 = adc.c3_out; /* Input reference to PID by ADC */
        pid.pid_fb_reg1  = adc.c4_out; /* Input feedback to PID by ADC */
}

void interrupt periodic_interrupt_isr() /* ISR function */
{
        pid.calc(&pid); /* Call compute function for PID */
        u=pid.pid_out_reg1; /* Access the output of PID */
}
```

[3]Não é necessário inicialização de *hardware*, já que se trata de módulo independete de *hardware*. Sendo assim só é invocado o método de cálculo a cada ocorrência da *ISR*.

3.5.5 Módulo PWM (Figura 3.15)

O módulo PWM usado opera em uma faixa de valores que vão de -32767 a +32767. Esses valores correspondem a um ciclo ativo de 100% e 0%, respectivamente.

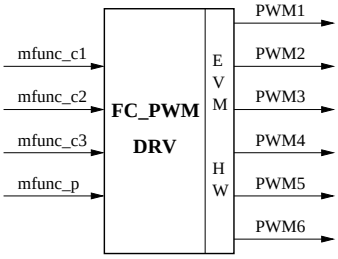

Figure 3.15: Módulo PWM[11]

Assim como os outros módulos é necessário a criação de uma estrutura chamada PWMGEN (Algoritmo 3.5) que contém as variáveis de confi guração, entrada e saída do PWM, sendo carregados seus valores iniciais com a atribuição de PWMGEN_DEFAULTS.

Algoritmo 3.5: Interface do módulo FC_PWM_DRV[11]

```
typedef struct
{
        int period_max;  /* PWM Period in CPU clock cycles */
        int mfunc_p;     /* Period scaler */
        int mfunc_c1;    /* PWM 1&2 Duty cycle ratio */
        int mfunc_c2;    /* PWM 3&4 Duty cycle ratio */
        int mfunc_c3;    /* PWM 5&6 Duty cycle ratio */
        int (*init)();   /* Pointer to the hardware init funtion */
        int (*update)(); /* Pointer to the update function */
} PWMGEN;
```

Após isto, para inicialização do *hardware* (já que se trata de um driver) invoca-se o método *init(&pwm)*. O período do PWM é regulado pela variável *period_max* e pode ser calculado como: $T = 2 * T1PR * \frac{1}{f_{clock}}$

O valor da variável *period_max* é atribuído ao registrador T1PR, que é o responsável pelo período de interrupção do *Timer*. Na aplicação foi adotado o período de 200μs (5KHz), correspondendo ao valor de 4000 no registrador T1PR.

O ciclo ativo do PWM gerado é ajustado pelo valor da variável *mfunc_c1*, que assume valores entre -32767 e 32767, por exemplo, para se obter um PWM com ciclo de 50% o valor da variável deve assumir 0. O método responsável pela atualização dos valores de saída do PWM é o *update(&gen)*.

O Algoritmo 3.6 exemplifi ca a utilização do Módulo PWM.

Algoritmo 3.6: Exemplo de utilização do módulo FC_PWM_DRV[11]

```
PWMGEN pwm = PWMGEN_DEFAULTS; /* Instantiation and Inicialization */

void main(void) /* Main function */
{
        pwm.period_max = 4000; /* Set the period to 4000 cycles */
        pwm.init(&pwm); /* Call the hardware initialization function */
}

void interrupt periodic_interrupt_isr()
{
        pwm.mfunc_c1 = pid.pid_out_reg1; /* Connect the PID to PWM */
        pwm.update(&pwm); /* Call the hardware update function */
}
```

3.5.6 Estrutura do programa exemplo

Seguindo os algoritmos apresentados já pode-se perceber uma ligação discreta entre os blocos. O algoritmo implementado funciona em função das interrupções geradas pelo *Timer*.

Inicialmente são instanciados e inicializados os módulos que serão utilizados (Algoritmo 3.7).

Algoritmo 3.7: Instanciação e Inicialização dos Módulos

```
/* Create data structures and load defaults values */
PWMGEN   pwm  = PWMGEN_DEFAULTS;
ADCVALS  adc  = ADC_DEFAULTS;
PIDREG1  pid  = PIDREG1_DEFAULTS;
WATCHDOG wdog = WATCHDOG_DEFAULTS;
```

A função *RstSystem()* (Algoritmo 3.8), chamada a partir da função (*main()*, Algoritmo 3.9), prepara o DSP para a aplicação, confi gurando registradores principais, habilitando e mascarando interrupções do *Timer*.

Algoritmo 3.8: Confi guração e inicialização do *DSP*

```
/*
* Reset system routine.
* First execute the initialization for the Wait Stage Genrator,
* Global interrupt disable, Shut off the Watchdog,
* and set up the Interupt Mask Register
*/
void RstSystem(void)
{
        /* Disable Wait-State Generator */
        WSGR = NO_WAIT_STATES;
```

```
        /* Disable interrupts */
        asm(" setc intm ");

        /* Watchdog shut off */
        wdog.disable();

        /* Configure the System Control and Statur Register 1 to initialize
        * the system and use the PWM timer (TIMER1 by EVA) to generate a
        * interrupt event.*/
        SCSR1=((EVA_CLKEN<<2)|ILLADR);

        /* Set up interrupt mask to enable INT2 */
        IMR = (INT2_MASK<<1);

        /* Set up interrupt mask to enable the Timer1 period interrupt */
        EVAIMRA = (T1PINT_ENABLE<<7);

        /* Enable interrupts */
        asm(" clrc intm ");

} /* End: RstSystem(void) */
```

Na função *main* do programa, o (Algoritmo 3.9), após inicializar o *DSP*, são configurados os módulos. É então escolhido quais canais do ADC serão utilizados e o conversor é inicializado. O mesmo acontece com o módulo PWM que recebe o período de atuação e é inicializado. Os parâmetros do controlador PID (*Kp*, *Ki*, *Kd*, etc.) também são ajustados nesta etapa.

Em seguida a interrupções são habilitadas e o programa entra em um laço infinito. A cada interrupção gerada pelo *Timer* (a cada 200μs), o programa é desviado à *ISR* correspondente, que neste caso foi denominada como *periodic_interrupt_isr()*.

Algoritmo 3.9: Inicialização dos módulos ADC/PWM/PID

```
/*
* Main program function
*/
void main(void)
{
        /* Reset system routine. */
        RstSystem();

        /* Init the ADC converter.
        * Configure the variables - channels as following:
        * c4 - Ch3 // c3 - Ch2 // c2 - Ch1 // c1 - Ch0 */
        adc.a4_ch_sel = 0x3210;
        adc.init(&adc);
```

```
        /* Init the PWM generator.  */
        pwm.period_max = 4000;
        pwm.init(&gen);

        /* Init the PID controller. */
        pid.Kp_reg1=10000;      /* Parameter: Proportional gain */
        pid.Ki_high_reg1=1500;  /* Parameter: Integral gain */
        pid.Ki_low_reg1=0;      /* Parameter: Integral gain */
        pid.Kd_reg1=0;          /* Parameter: Derivative gain */
        pid.pid_out_max=-20000; /* Parameter: Maximum PID output */
        pid.pid_out_min=-32000; /* Parameter: Minimum PID output */

        /* Enable interrupts */
        asm(" clrc intm ");

        /* Nothing running in the background at present */
        while(1)
        {
        }
} /* End: main() */
```

Realizado as inicializações os módulos já estão prontos para serem conectados. A conexão destes é feita na rotina de tratamento da interrupção (*periodic_interrupt_isr*, Algoritmo 3.10). A periodicidade desta rotina é dada pela ocorrência da interrupção gerada pelo *timer* que é mesmo período do PWM já que foi utilizado o mesmo *timer* usado por ele. Poderia ter sido utilizado outro *timer* do *DSP*, mas isso poderia causar problemas pois teria-se o acontecimento de duas interrupções que poderiam se sobrepor, como não era escopo deste trabalho o desenvolvimento de um escalonador para manipular as interrupções resolveu-se simplificar a aplicação.

3.5.6.1 Conectando os módulos

A rotina de controle começa com a atualização dos valores convertidos pelo ADC, feito pelo método adc.update(&adc), lembrando que são adquiridos os sinais referentes à referência e à medição da velocidade do motor. Após a conversão as variáveis *c4_out* (referência) e *c3_out* (engfeedback, velocidade do motor no instante) são fornecidas como parâmetros de entrada ao controlador PID, através da seguinte atribuição:

```
        pid.pid_fb_reg1  = adc.c3_out;
        pid.pid_ref_reg1 = adc.c4_out;
```

Basta executar o método *calc(&pid)* e ação de controle é calculada passando a figurar como a variável *pid_out_reg1*. O valor desta está compreendido entre uma faixa de valores que são fornecidos ao controlador (*pid_out_max* e *pid_out_min*), correspondendo à saturação da ação de controle. Foram

identificados que valores de ciclo ativo correspondiam a máxima e mínima velocidades do motor, determinando a saturação do controlador. O próximo passo consiste em codificar a ação de controle calculada pelo controlador PID em um valor de ciclo ativo para o PWM, isto foi feito diretamente ligando as variáveis correspondentes:

```
        pwm.mfunc_c1 = pid.pid_out_reg1;
```

É então executado o método *update(&pwm)*, determinando assim o valor de ciclo ativo que só será modificado na próxima amostragem. O código completo da *ISR* pode ser visto no Algoritmo 3.10.

Algoritmo 3.10: Rotina de tratamento da interrrupção

```
/*
* Interrupt Service Routine (ISR): Timer2 interrupt
*/
void interrupt periodic_interrupt_isr(void)
{
        /* Disable interrupts */
        asm(" setc intm ");

        adc.update(&adc);

        /* PID Controller */
        pid.pid_ref_reg1 = adc.c4_out;
        pid.pid_fb_reg1  = adc.c3_out;
        pid.calc(&pid1);

        /* Generate PWM by PID output */
        pwm.mfunc_c1 = pid.pid_out_reg1;
        pwm.update(&pwm);

        /* Enable interrupts */
        asm(" clrc intm ");

} /* End: periodic_interrupt_isr() */
```

3.6 Conclusão

Este capítulo descreveu o *meta-framework* proposto pela TI com o intuito facilitar o desenvolvimento de sistemas embutidos utilizando processadores digitais de sinais (*DSPs*). Pôde-se ter uma idéia da gama de técnicas que são utilizadas para o desenvolvimento eficiente de sistemas, desde metodologias que podem ser adotadas no projeto e codificação até as características importantes que um *kernel* de tempo real traz para aplicações multitarefas.

Pretendeu-se com este capítulo mostrar a base da qual partimos para o desenvolvimento do nosso *framework* e também fazer um levantamento do que existe para não incorrer no típico erro de 'reinventar a roda'.

A informação contida neste capítulo está espalhada por dezenas de documentos da TI, os quais nem sempre são consistentes entre si. Procurou-se portanto, definir um conjunto consistente de termos, embora isto signifique uma certa liberdade em relação à alguns documentos da TI, especialmente os mais antigos.

Capítulo 4

Framework proposto

Apresentar-se-á neste capítulo uma extensão do *framework* da Texas Instruments para projeto de sistemas utilizando diagrama de blocos[8]. A principal alteração será evoluir o atual *framework*, restrito a um modelo monotarefa, para um modelo multitarefa. Serão descritas as adaptações feitas e as inclusões necessárias para que essa proposta se torne válida.

Com esta extensão pretende-se disponibilizar ao engenheiro de controle um modelo de desenvolvimento de sistemas embutidos que, além de mapear um diagrama de blocos para um diagrama de componentes de *software*, permita a existência de múltiplas tarefas concorrendo entre si, sem que seja necessário uma mudança signifi cativa de paradigma de projeto.

4.1 Introdução

O *framework* para o desenvolvimento de sistemas utilizando diagramas de blocos, proposto pela Texas Instruments (Capítulo 3), permite que pela simples e intuitiva conexão entre componentes de *software*, representações algorítmicas de blocos funcionais, se possa construir diversos tipos de aplicações, tendo como ponto forte o reuso dos componentes já desenvolvidos. Este *framework* é baseado em um modelo monotarefa, onde uma série de operações são realizadas de modo seqüencial até o fi m da tarefa (Algoritmo 3.10), sem que haja interrupção em sua execução.

O modelo monotarefa é amplamente utilizado, e indicado, quando se possui um *hardware* totalmente dedicado a uma única e determinada tarefa, como controlar o motor de um refrigerador. Usualmente, a execução da tarefa é dirigida pela ocorrência de uma interrupção, normalmente do *timer*, em intervalos periódicos.

Mas muitas aplicações reais são compostas por mais de uma tarefa, até por pressões do mercado que quer produtos mais atrativos com mais funcionalidades, como a existência de uma interface humano-máquina além do controlador, o que torna este modelo muitas vezes de difícil utilização.

Uma forma de resolver este problema é estender o modelo monotarefa da TI para um modelo multitarefa, onde as diversas funcionalidades podem ser divididas em tarefas que concorrem entre si.

No modelo multitarefa proposto manteve-se o padrão de desenvolvimento de componentes (*Algorithm Standard*[3][13]), inseriu-se uma nova simbologia para tratar a multitarefa e criou-se estruturas adicionais para suportar a comunicação entre tarefas. Também será necessário um suporte de execução apropriado, na forma de um *microkernel*, o qual será descrito no Capítulo 5.

4.2 Componentes de *Software*

O projeto de sistemas baseado em diagramas de blocos é bastante intuitivo, conhecido pelo engenheiro de controle e utilizado em uma grande quantidade de ferramentas para controle de processos. O mapeamento destes blocos funcionais para componentes de *software* [8] torna esta metodologia ainda mais forte criando um canal confiável de comunicação entre o engenheiro de controle e o engenheiro de *software*.

O bloco funcional representa, resumidamente, um modelo matemático de um processo ou, em muitas vezes, de uma parte do processo. Este modelo matemático realiza cálculos sobre valores de entrada gerando valores de saída, como pode ser visto no bloco genérico ilustrado na Figura 4.1.

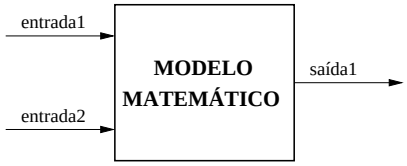

Figura 4.1: Bloco Funcional genérico.

No mapeamento de um bloco funcional para um componente de *software* as entradas e saídas são atributos do componente e o modelo matemático corresponde a um algoritmo de computação. O componente genérico ilustrado pela Figura 4.2 mapea o bloco genérico criado. Pode se perceber a grande semelhança externa entre as duas figuras e isso cria grande motivação em realizar a conversão de um bloco funcional para um componente de *software*.

A partir deste mapeamento e seguindo as especificações formais do *Algorithm Standard* tem-se como resultado a geração de um algoritmo genérico que poderia ser aplicado a qualquer componente

Figura 4.2: Componente de *Software* genérico.

(Algoritmo 4.1). De forma a tornar a geração de algoritmos padronizados, determinou-se que todos os componentes possuam operações de configuração (*init*) e de computação (*update*), mesmo que alguma dessas operações venha, em razão do tipo de componente, a não fazer nada (vazio).

Algoritmo 4.1: Algoritmo de um componente genérico

```
/*-----------------------------------------------------------------------------
 Define a estrutura do COMPONENTE_GENERICO
-----------------------------------------------------------------------------*/
typedef struct {
   int entrada1;
   int entrada2;
   int saida1;
   int (*init)();
   int (*update)();
   } COMPONENTE_GENERICO;

/*-----------------------------------------------------------------------------
 Define um ponteiro para COMPONENTE_GENERICO
-----------------------------------------------------------------------------*/
typedef COMPONENTE_GENERICO  *COMPONENTE_GENERICO_handle;

/*-----------------------------------------------------------------------------
 Inicializacao padrao para o objeto COMPONENTE_GENERICO
-----------------------------------------------------------------------------*/
#define COMPONENTE_GENERICO_DEFAULTS {NULL,     \
                                      NULL,     \
                                      NULL,     \
                                      (int (*)(int))COMPONENTE_GENERICO_Init,  \
                                      (int (*)(int))COMPONENTE_GENERICO_Update \
                                      }

/*-----------------------------------------------------------------------------
 Prototipos das funcoes
 -----------------------------------------------------------------------------*/
void COMPONENTE_GENERICO_Init(COMPONENTE_GENERICO_handle);
void COMPONENTE_GENERICO_Update(COMPONENTE_GENERICO_handle);

/*-----------------------------------------------------------------------------
 Funcao de inicializacao do Componente
-----------------------------------------------------------------------------*/
void inline COMPONENTE_GENERICO_Init(COMPONENTE_GENERICO *p)
{
/* CONFIGURACAO INICIAL */
}

/*-----------------------------------------------------------------------------
 Funcao de computacao/atualizacao do Componente
-----------------------------------------------------------------------------*/
```

```
void inline COMPONENTE_GENERICO_Update(COMPONENTE_GENERICO *p)
{
/* ALGORITMO DE COMPUTACAO */
}
```

As representações gráficas para os vários tipos de componentes de acordo com a sua dependência já foram amplamente descritas no Capítulo 3 e o *framework* proposto mantém a mesma representação para todos os componentes.

4.3 O Modelo Multitarefa

Na grande maioria dos sistemas pode-se observar uma grande quantidade de tarefas executando em paralelo. Mesmo nos sistemas embutidos é possível perceber a existência de mais de uma tarefa, controle de um motor aliado à uma interface humano-máquina com comunicação serial ou rede local, por exemplo. Nestes casos, o modelo monotarefa não consegue ser aplicado e adaptações simples criam problemas como atraso da tarefa. Isso porque o modelo é direcionado para a execução de uma tarefa e não de várias.

Para atender um sistema que necessita de várias tarefas executando deve-se utilizar um modelo multitarefa. Este modelo permite a existência de diversas tarefas concorrendo entre si. A utilização deste modelos com um suporte adequado (assunto do Capítulo 5), onde as tarefas podem receber prioridades e tempos de atendimento individuais, com requisitos temporais (tarefas com maior prioridade podem interromper as de menor prioridade) torna este modelo muito atrativo. Com ele, aumenta-se o processamento de tarefas por unidade de tempo (*throughput*) além de diminuir os tempos de resposta.

Para aplicar o modelo multitarefa sobre a metodologia de desenvolvimento antiga precisou-se criar um elemento para delimitar as fronteiras entre as tarefas e para isso utilizou-se um retângulo tracejado com cantos arredondados. Cada tarefa deve ser envolvida por este delimitador.

O papel de criação de um sistema embutido é um processo no qual é necessária a interação entre o engenheiro de *hardware* e o engenheiro de *software*, que normalmente não são as mesmas pessoas. Para tanto, incluiu-se no modelo outro símbolo, e que faz parte da tarefa, para representar o *hardware* externo conectado ao componente de *software*: um retângulo tracejado com cantos aguçados. Isso possibilita ao engenheiro de *software* criar componentes específicos para um determinado dispositivo periférico (*device driver*). Por exemplo, na Figura 4.3 o bloco “MOTOR” é um exemplo desta classe, pois se trata de um *hardware* externo conectado a um periférico programado em *software*.

Supondo um sistema com três tarefas: controlar um motor e oferecer ao usuário uma interface de comunicação com recebimento e envio de dados, considerando componentes já desenvolvidos e a simbologia acima especificada, teríamos um sistema multitarefa ilustrado pela Figura 4.3.

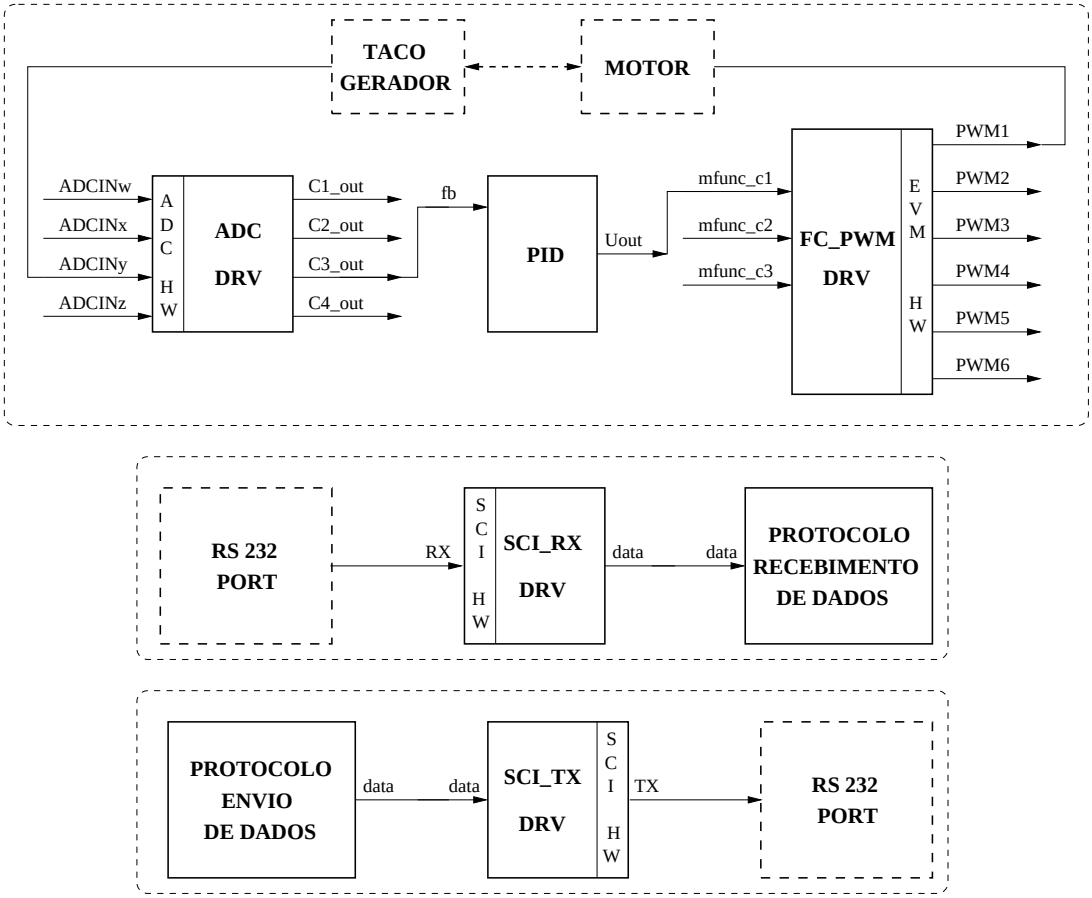

Figura 4.3: Modelo Multitarefa.

O código de conexão entre os blocos já foi descrito detalhadamente no Capítulo 3 e será novamente descrito no Capítulo 6 para o *framework* proposto. Sucintamente as três tarefas gerariam o seguinte código genérico, individualmente:

```
void tarefa_generica(void)
{
        while(1)
        {
                /* CODIGO DA TAREFA:
                 * - CONEXAO ENTRE OS BLOCOS E SUPORTE OPERACIONAL ADEQUADO
                 */
        }
}
```

Mas um sistema multitarefa não possui, simplesmente, tarefas executando independentemente. O sistema ilustrado tão pouco vai fazer sentido em realizar o recebimento de dados se não for para configurar os parâmetros do controlador ou o envio de dados se não for para visualizar o comportamento

do controlador. Para isso é necessário que as tarefas se comuniquem e isto será abordado na seção seguinte.

4.4 Comunicação entre tarefas

A comunicação entre tarefas é uma importante característica que deve ser considerada em sistemas multitarefa, isso porque para realizar a comunicação são utilizados recursos que são compartilhados entre todas as tarefas e num sistema multitarefa não deve-se permitir o acesso a um mesmo recurso por duas tarefas ao mesmo tempo, seja para leitura e/ou escrita (exclusão mútua).

Nos sistemas computacionais multitarefa existem, basicamente, duas maneiras de se implementar esta comunicação: memória compartilhada e troca de mensagens.

Na memória compartilhada o núcleo operacional deve fornecer primitivas que possibilitem a sincronização entre tarefas (semáforos e/ou monitores, por exemplo), isso como forma de manter a integridade dos dados compartilhados, não só protegendo-os como também sincronizando tarefas de escrita e leitura.

Já na troca de mensagens o sistema deve fornecer primitivas de maior abstração e o acesso aos dados compartilhados é feito, normalmente, dentro do núcleo do sistema. No *framework* proposto optou-se por utilizar a troca de mensagens devido ao próprio modelo de componentes que sugere uma maior abstração na manipulação ds dados compartilhados.

Existem, resumidamente, dois tipos de mecanismos que a comunicação por troca de mensagens pode se utilizar: mecanismos síncronos e mecanismos assíncronos.

No mecanismo de comunicação síncrono sempre que duas tarefas querem se comunicar elas devem se sincronizar através da troca de mensagens, este processo é chamado de *rendezvous*. Assim, se o emissor inicia primeiro ele deve esperar até que o receptor receba a mensagem, por outro lado, se o receptor iniciar primeiro este deve esperar até que o emissor produza a mensagem.

Em um mecanismo de comunicação assíncrono as tarefas não precisam esperar uma pela outra. O emissor deposita a mensagem dentro de um canal de comunicação e continua sua execução, independente da condição do receptor. Similarmente, assumindo que a última mensagem foi depositada dentro do canal de comunicação, o receptor pode acessar diretamente a mensagem sem a necessidade de sincronização com o emissor.

Em sistemas com restrições temporais os mecanismos síncronos podem causar atrasos na execução das tarefas fazendo estas perderem seus *deadlines*, além de que estes mecanismos podem ocasionar inversões de prioridade e bloqueios indefinidos na execução de uma tarefa, causando um comportamento temporal imprevisível.

Apesar do mecanismo assíncrono não garantir que o receptor tenha recebido a mensagem, percebe-se que ele torna o mecanismo mais flexível temporalmente no sentido que a tarefa continua sua execução cumprindo com seu *deadline*.

Devido a necessidade das aplicações de controle possuírem restrições temporais para a execução das tarefas no *framework* proposto foram especificados dois tipos abstratos de dados para comunicação assíncrona entre elas, conforme a necessidade de utilização. Cada tipo abstrato possui sua simbologia e apresenta diferentes características.

4.4.1 Tipo abstrato FILA

A FILA é um tipo abstrato criado para ser utilizado quando da necessidade de armazenamento de amostragens de um determinado elemento. É utilizada, normalmente, na comunicação entre uma tarefa de controle e uma tarefa de emissão de dados, mas pode ser utilizada entre duas tarefas quaisquer quando existe a necessidade de se possuir um histórico de um determinado elemento.

É definido por uma estrutura de dados especificada no núcleo de suporte (Capítulo 5), representada pela Figura 4.4, onde a entrada é realizada por uma operação de inserção (*put*) e a saída por uma operação de retirada (*get*) de elementos. Este bloco é baseado em conceito de *buffer* circular do tipo FIFO (*First In First Out*).

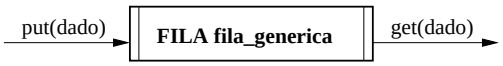

Figura 4.4: Tipo abstrato FILA.

O código gerado por seu componente (abaixo) resulta em um apontador do tipo FILA onde a estrutura de dados será criada por primitiva do núcleo de suporte, onde a capacidade da fila é definida pelo usuário.

```
FILA *fila_generica;
```

4.4.2 Tipo abstrato ATRIBUTO

O tipo abstrato ATRIBUTO foi criado com o objetivo de conter atributos (parâmetros) de configuração ou de armazenamento em que somente o último valor interessa. É utilizado quando se tem a necessidade de armazenar valores únicos de um determinado parâmetro.

Possui uma representação gráfica similar ao tipo FILA mas, diferentemente dele, no tipo ATRIBUTO as entradas realizam operações de atribuição (*set*) e as saídas operações de leitura (*get*) de

elementos[1],como pode ser visto na Figura 4.5.

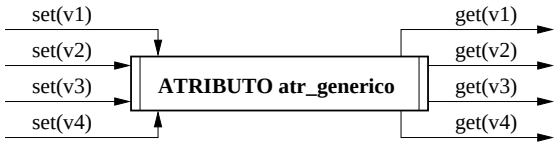

Figure 4.5: Tipo abstrato ATRIBUTO.

A estrutura de dados do tipo ATRIBUTO é definida pelo usuário que estipula os campos que farão parte do tipo de dado, conforme pode ser visto no código gerado abaixo. Assim como o tipo FILA, é declarado um apontador para o tipo definido e que será criado por primitiva do núcleo de suporte.

```
typedef struct
{
        int v1;
        int v2;
        int v3;
        int v4;
} ATRIBUTO_GENERICO;

ATRIBUTO_GENERICO *atr_generico;
```

4.5 Conclusão

Neste capítulo foi proposto um *framework*, implementado sobre um modelo multitarefa, como uma alternativa ao modelo monotarefa do *framework* da Texas Instruments (Capítulo 3).

Teve-se por objetivo mostrar os benefícios e facilidades de um modelo multitarefa, em relação ao modelo monotarefa, com o cuidado em mantê-lo compatível com conceitos já definidos anteriormente. Foram descritos e ilustrados os diversos elementos necessários à implementação deste *framework* e sempre que necessário o código gerado por cada elemento foi listado.

No Capítulo 5 será descrito o μKernel desenvolvido para suportar o *framework* proposto. E a aplicação deste em um sistema multitarefa real, onde os elementos poderão ser visualizados conectados a tarefas individuais ou na aplicação como um todo, são assunto do Capítulo 6.

[1]Para representação de grande quantidade de parâmetros pode ser utilizado uma única entrada (*put*) ou saída (*get*) com todos os elementos utilizados. Por exemplo: *put(v1,v2,v3,v4)* e *get(v1,v2,v3,v4)*

Capítulo 5

Descrição do μKernel: API, projeto e codificação

Este capítulo apresentará as características de projeto e codificação de um *microkernel* multitarefa de tempo real (μKernel) utilizado como suporte para o desenvolvimento de aplicações de controle embutidas em DSP. O μKernel descrito neste capítulo foi implementado para o processador TMS320C2407A da Texas Instruments.

É importante notar que o μKernel foi construído de forma a suportar o *framework* desenvolvido por nós e que está explicado no capítulo anterior.

5.1 Introdução

Um *kernel* representa a parte mais interna de um sistema operacional e está ligado diretamente ao *hardware* da máquina, provendo assim uma abstração que isola a aplicação dos inconvenientes gerados no acesso a registradores e dispositivos de *hardware*.

Usualmente, um *kernel* realiza as seguintes atividades básicas:

- Gerenciamento de processos;
- Manipulação de interrupções;
- Sincronização entre processos.

Gerenciamento de processos é o serviço mais básico que um sistema operacional pode prover. Ele inclui várias funções de suporte como: criação e finalização de processos, escalonamento e despacho de tarefas, chaveamento de contexto, entre outras.

O objetivo do mecanismo de manipulação de interrupções do *kernel* é prover serviço, através da execução de uma rotina de tratamento dedicada, àquelas requisições de interrupção geradas por um dispositivo periférico que é utilizado diretamente por ele, por exemplo a utilização da interrupção do *timer* para manipular o tempo e controlar o período das tarefas. Em outras requisições de interrupção como: teclado, porta serial, conversores analógicos digitais, etc, o serviço é provido por controladores de dispositivo específicos (*device driver*) aos quais a interrupção do dispositivo em questão é associada e que realizam operações de leitura e/ou escrita. O controlador é responsável por traduzir estas operações em uma seqüência de acionamentos eletrônicos, elétricos e mecânicos capazes de realizar a solicitação [17].

Em sistemas operacionais de propósito geral as tarefas podem sempre ser interrompidas pelos controladores, a qualquer momento. Entretanto, em sistemas de tempo real, esta abordagem pode introduzir esperas imprevisíveis na execução de tarefas críticas, podendo ocasionar a perda de algum *deadline* importante. Por esta razão, em um sistema de tempo real, o mecanismo de manipulação de interrupções deve estar integrado com o mecanismo de escalonamento, assim um controlador deve ser escalonado como qualquer outra tarefa do sistema.

Outro importante serviço fornecido pelo *kernel* é um mecanismo que suporte sincronização e comunicação entre tarefas. Em sistemas operacionais de propósito geral isto é feito através de semáforos, os quais apresentam uma solução eficiente para o problema de sincronização, bem como para o problema de exclusão mútua. Entretanto semáforos fazem com que o sistema fique propenso a sofrer inversão de prioridades, bloqueando a execução de tarefas e não garantindo a temporalidade do sistema. Como conseqüência, de forma a garantir a previsibilidade, um *kernel* de tempo real fornece tipos especiais de semáforos que suportam um protocolo de acesso ao recurso (*Priority Inheritance*, *Priority Ceiling* ou *Stack Resource Policy*) para que não ocorram inversões de prioridade [4].

A quantidade de serviços oferecidos por um *kernel* pode ser bastante vasta, desde o simples chaveamento de contexto até a criação dinâmica de tarefas incluindo um gerenciamento de memória através de coletores de lixo (*garbage colectors*). O que determina quais serviços serão oferecidos pelo *kernel* são as restrições da plataforma alvo (como: memória, capacidade de processamento, etc.) e as restrições da aplicação (como: multitarefa, comunicação entre tarefas, etc.). Avaliadas estas características pode-se levantar os requisitos necessários ao desenvolvimento do *kernel*.

5.2 Arquitetura da plataforma alvo

Levando-se em consideração todo estudo desenvolvido sobre a tecnologia da Texas Instruments e a disponibilidade de kits doados pela empresa ao departamento, achou-se como melhor alternativa trabalhar com o processador digital de sinais (DSP) da Texas Instruments como o alvo do desenvolvimento do μKernel. Toda a teoria aqui discutida é aplicável em qualquer tipo de processador ou controlador desde que se mantenha fiel ao *framework* especificado e ao modelo de componentes.

O DSP utilizado no desenvolvimento do μKernel foi o TMS320C2407A embutido em uma placa de desenvolvimento chamada eZdsp acompanhada de um ambiente integrado de desenvolvimento (*Code Composer*) com compilação, depuração e emulação.

O TMS320C2407A é um controlador DSP da família C24x e sua arquitetura é mostrada na Figura 5.1. Sua arquitetura é baseada em uma arquitetura *Harvard* modificada composta por dois barramentos separados, um para programa e outro para dados. Ainda possui um terceiro barramento para os periféricos embutidos no controlador (conversores analógicos digitais, interface de comunicação serial, etc.) mas que são mapeados no espaço do barramento de dados através de um módulo especial do sistema. Assim, todas as instruções que operam no espaço de dados também operam sobre todos os registradores dos periféricos.

A separação entre o barramento de dados e o barramento de programa permite acesso simultâneo à instruções de programa e dados. Por exemplo, enquanto um dado é multiplicado um produto anteriormente armazenado pode ser somado ao valor do acumulador e ao mesmo tempo um novo endereço pode ser gerado. Tal paralelismo suporta um conjunto de operações aritméticas, lógicas e de manipulação de bits podendo todas serem realizadas em um único ciclo de máquina. O processador também inclui mecanismos de controle para gerenciar interrupções, operações repetidas e chamadas a subrotinas.

5.2.1 Memória

O TMS320C2407A contém as seguintes configurações de memórias embutidas:

- *Dual-access random-access memory* (DARAM) - com 544 *words* x 16 *bits* nos quais é possível acessar para escrita e leitura em um mesmo ciclo de máquina (*pipeline*). Esta memória é o endereço principal para manter dados mas, quando necessário, pode também manter programas;
- *Single-access random-access memory* (SARAM) - tem 2k *words* x 16 *bits* que, ao contrário da DARAM, só podem ser acessados para escrita ou leitura em um ciclo de máquina, mas assim como a DARAM pode mapear o espaço de dados, de programa ou ambos;
- *Flash* EEPROM - provê uma atrativa alternativa para mascarar programas (permitir somente acesso a leitura). Como em uma memória ROM (*Read Only Memory*) a *Flash* não é volátil entretanto tem a vantagem de ser reprogramável no próprio dispositivo (EEPROM - *Electrically Erasable Programmable ROM*). Este DSP incorpora 32K *words* x 16 *bits* de *Flash* para espaço de programa. O módulo *Flash* tem múltiplos setores que podem ser individualmente protegidos enquanto são apagados ou programados, o tamanho deles não é uniforme e está particionado em setores com 4K/12K/12K/4K.

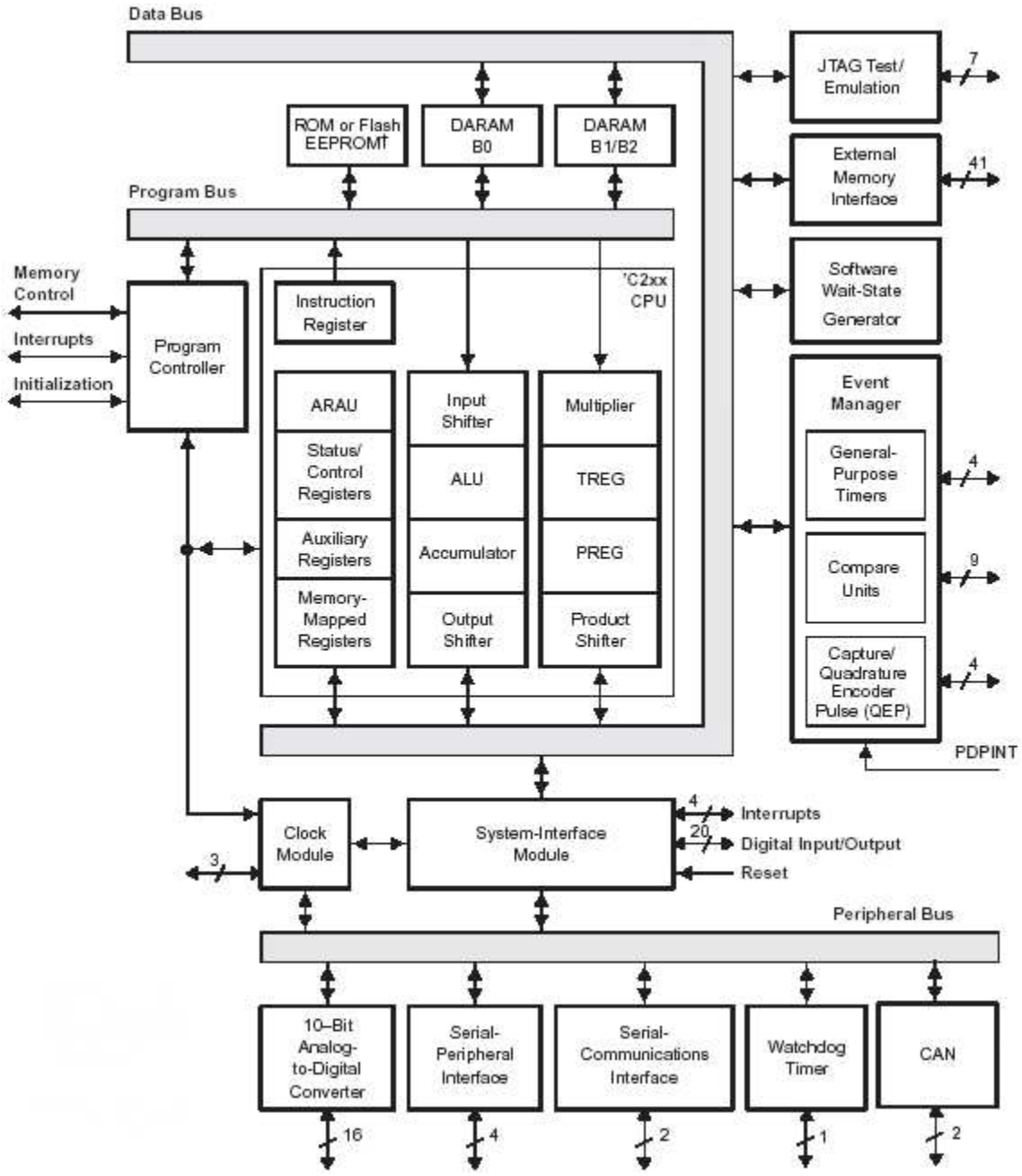

Figura 5.1: Diagrama de Blocos Funcionais do DSP utilizado [12]

- ROM - memória somente de acesso a leitura que está mascarada no espaço de programa. Esta memória é programada diretamente com o fabricante e geralmente são utilizadas quando o código de uma aplicação está estável e a demanda por esta é tão grande que justifique não gravar dispositivo por dispositivo (produção em larga escala).
- *Boot* ROM - É uma memória do tipo ROM de 256 *words* que vem pré programada de fábrica e que contém um programa monitor que é carregado quando o dispositivo sofre um *reset* e possui um determinado pino externo conectado em 'terra'. Este programa monitor é responsável por inicializar a comunicação serial do DSP e gravar os dados enviados em sua memória *flash*, isso torna possível programar o DSP via uma interface serial.

Os diversos tipos de memória citados estão referenciados no mapa de memória e ilustrados na Figura 5.2, com seus respectivos setores divididos. Ao total a memória possui:

- 64K *words* para programa;
- 64K *words* para dados locais;
- 32K *words* para dados globais;
- 64K *words* para entrada/saída (periféricos e memória externa);

5.2.2 Unidade Central de Processamento (CPU)

A CPU do TMS320C2407A é a mesma de todos os dispositivos da família C2xx da Texas Instruments e contém:

- Uma unidade central lógica e aritmética (CALU) de 32 *bits* - a CALU realiza operações utilizando aritmética de complemento de 2, pode utilizar dados vindos da memória (16 *bits*) ou resultado de 32 *bits* do multiplicador. A saída da CALU é armazenada no acumulador;
- Um acumulador de 32 *bits*;
- Um multiplicador de 16 *bits* x 16 *bits* que gera um resultado de 32 *bits*. Utilizado eficientemente o DSP realiza operações fundamentais para o processamento digital de sinais como convoluções, correlações e filtragem;
- Lógica de geração de endereços de dados, a qual inclui oito registradores auxiliares e uma unidade auxiliar de aritmética (ARAU);
- Lógica de geração de endereçamento de programa.

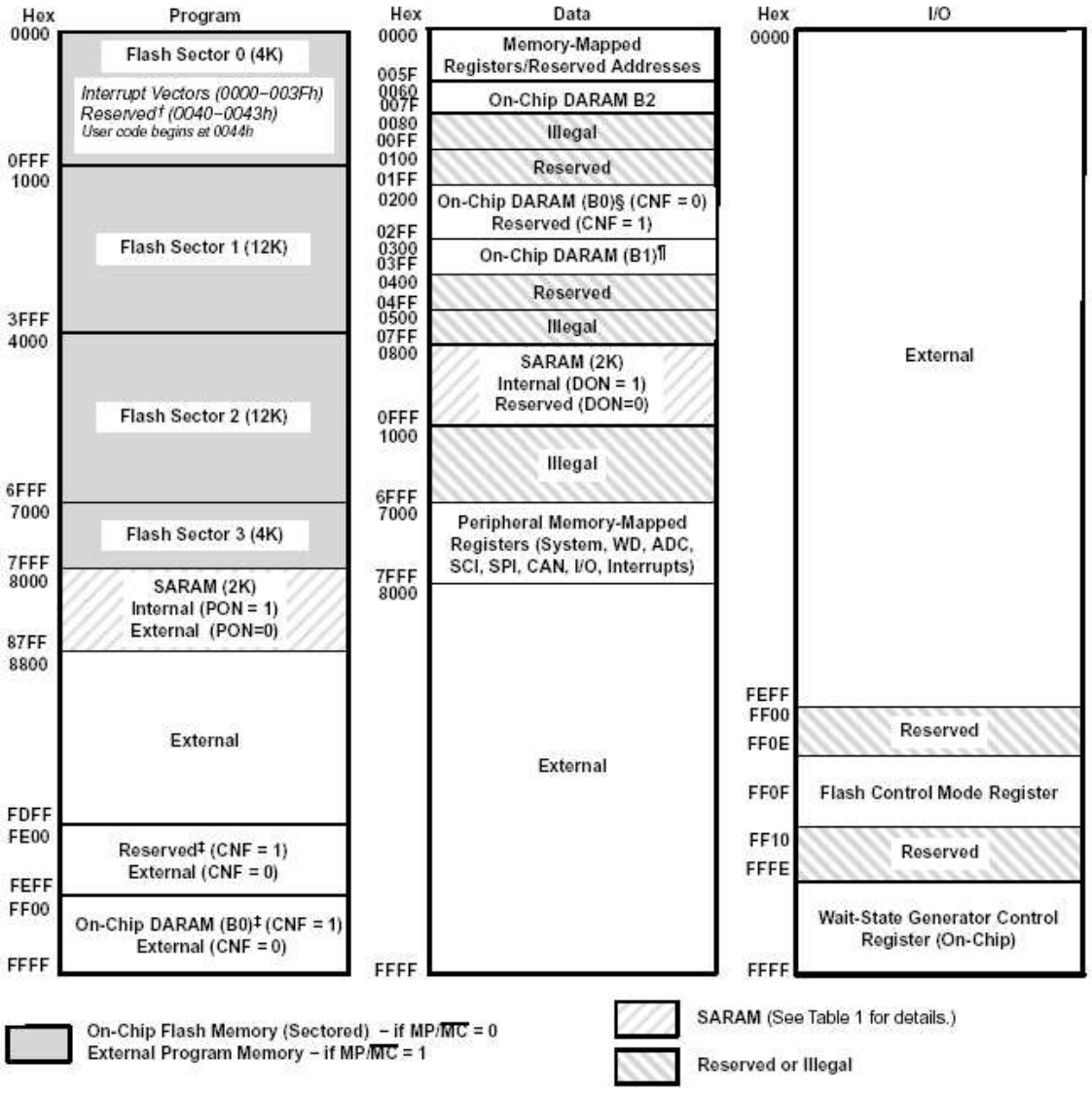

Figure 5.2: Mapa de memória do DSP utilizado [12]

5.2.3 Periféricos

Se tratando de um controlador DSP para motores sua grande característica, além da arquitetura do processador é a grande variedade de periféricos embutidos. São os seguintes módulos:

- Gerenciadores de eventos (EVA, EVB) - inclui 2 módulos que contém: *timers* de propósito geral, unidades de comparação e geração de PWM (*Pulse Width Modulation*), unidades de captura e circuitos para geração de pulsos de onda quadrada.
- Conversores analógico para digital (ADC) - este módulo consiste em 16 conversores de 10 *bits* com um circuito de *sample-and-hold* (amostra e mantém) embutido.
- Um módulo para rede de controladores (CAN) - implementando o protocolo para comunicação com controladores ou equipamentos que utilizem o mesmo protocolo.
- Uma interface para comunicação serial (SCI) - suporta comunicações digitais entre a CPU e outros periféricos assíncronos que utilizam o mesmo padrão de formato.
- Uma interface para comunicação com periféricos (SPI) - comunicação digital síncrona, normalmente utilizada para comunicações entre o controlador DSP e periféricos externos (*display*, ADC externo) ou outro processador. Comunicações entre vários dispositivos utiliza o método de operação mestre/escravo.
- Portas de entrada/saída digitais com funções compartilhadas - possui 41 portas digitais de propósito geral, bidirecionais e em sua grande maioria compartilhadas com periféricos.
- Interface de memória externa - pode endereçar mais de 64K x 16 *words* de memória em cada um dos espaços de dados, programa ou entrada/saída.
- Cão de guarda (*Watchdog* - WD) - a função deste módulo é monitorar as operações de *software* e *hardware* gerando uma reinicialização do sistema se não for periodicamente reiniciado por *software*, através da escrita de uma determinada chave.

Foram inseridos aqui os itens que, superfi cialmente, dão uma noção da plataforma alvo do μKernel. Maiores detalhes podem ser encontrados em documentos técnicos específi cos disponíveis no próprio sítio da Texas Instruments na *web* (www.ti.com).

5.3 Requisitos do μKernel

O *framework* proposto no capítulo 4 teve como objetivo estender o modelo de trabalho proposto pela Texas Instruments, com isso foram introduzidos elementos que permitissem características como: execução de múltiplas tarefas e comunicação entre elas.

Para que esse novo modelo pudesse ser suportado se fez essencial a criação de um *microkernel* de tempo real (μKernel) que possuísse os serviços essenciais de forma a não sobrecarregar o processador e que assim fosse pequeno (poucas linhas de código) o suficiente para não ocupar muita memória na plataforma alvo. Existiu, também, a preocupação de que o μKernel pudesse receber componentes desenvolvidos conforme o *meta-framework* da Texas Instruments, que abrange técnicas para a padronização de codificação de elementos de forma a poderem ser intercambiáveis entre desenvolvedores.

Levando em consideração as restrições envolvidas no desenvolvimento do μKernel, estipulou-se os serviços essenciais a serem prestados por este:

- Gerenciamento de tarefas - serviços disponibilizados para manutenção e execução das tarefas:
  - criação de tarefas;
  - escalonamento e despacho de tarefas;
  - chaveamento de contexto.
- Manipulação do tempo: serviço que trata das requisições de interrupção geradas pelo *timer* do DSP.
- Comunicação entre tarefas - serviços de troca de dados ou configuração de componentes:
  - tipo abstrato FILA;
  - tipo abstrato ATRIBUTO.

No restante deste capítulo será descrita a sua estrutura, sua API (serviços visíveis aos seus usuários) do núcleo de tempo real proposto (μKernel), o funcionamento de seus mecanismos internos e como são realizadas as comunicações entre tarefas.

5.4 Estrutura do μKernel

As várias funções do μKernel estão organizadas conforme a estrutura hierárquica ilustrada na Figura 5.3. Esta estrutura pode ser dividida em quatro camadas:

- Camada de máquina: esta camada interage diretamente com o *hardware* da máquina. As primitivas realizadas por este nível são atividades fundamentais e não são visíveis ao usuário;
- Camada de gerenciamento de estados das tarefas: para manter os estados (pronto, executando, etc.) de diversas tarefas, o *microkernel*, precisa gerenciar algumas listas onde as tarefas estarão agrupadas por estado. Esta camada fornece primitivas para inserção e remoção de tarefas para as listas;

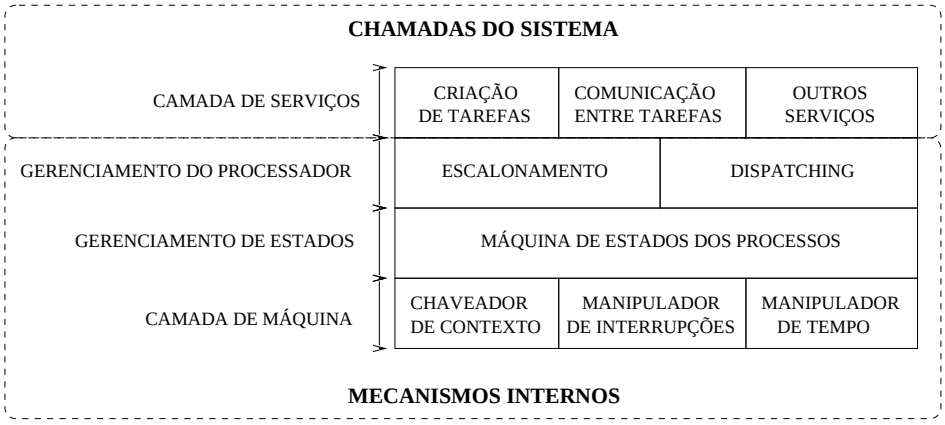

Figure 5.3: Estrutura hierárquica do μKernel

- Camada de gerenciamento do processador: os mecanismos desenvolvidos nesta camada somente dizem respeito às operações de escalonamento e despacho de tarefas;
- Camada de serviços: Este nível fornece todos os serviços visíveis ao usuário, com um conjunto de chamadas de sistema, este conjunto de serviços é denominado protocolo de interface da aplicação ou API. Alguns serviços a serem fornecidos são: criação de tarefas, mecanismos de comunicação entre tarefas, entre outros.

5.5 API do μKernel

O protocolo de interface da aplicação ou API (*Application Protocol Interface*) compreende as primitivas do μKernel disponíveis publicamente, isto é, os serviços que podem ser utilizados pelo usuário do μKernel. Estas primitivas são as chamadas do sistema operacional e formam a camada de serviços.

5.5.1 Camada de serviços

- **void init_system(unsigned int** *tick***)**, inicialização do μKernel. Após executada esta operação o programa principal se torna uma tarefa não tempo real na qual serão criadas as novas tarefas concorrentes. As atividades mais importantes executadas pelo *init_system* são:
  - Inicialização das filas de estado dos processos;
  - Configuração dos registradores de interrupção;
  - Preparação do descritor associado à tarefa *main*;

  - Configuração do período do *timer* para corresponder ao *tick* do sistema, que é dado por $f_{cpu}/(20000 * tick)$, o que quer dizer que um *tick* unitário faz com que o sistema gere interrupções a cada 1 *ms* ou $1 * 10^{-3}$ segundos, isso considerando um processador com f_{cpu} = 20 MHz (freqüência do processador que está sendo utilizado).

- **void create_task(void** $*tarefa$**, int** $tipo$**, int** *periodo***, int** *prioridade***)**, esta primitiva do μKernel reserva espaço e inicializa todas as estruturas de dados necessárias à tarefa colocando-a logo após na lista de tarefas em ESPERA. Os parâmetros de entrada descrevem:

  - $*tarefa$ - endereço inicial da tarefa.
  - $tipo$ - configura se a tarefa é tempo real ou não tempo real (definidos por HARD ou NRT).
  - *período* - se a tarefa for HARD associa o período estipulado à tarefa que está sendo criada. O período é definido em função da quantidade de *ticks* decorridos até a próxima ativação da tarefa.
  - *prioridade* - define a importância da tarefa no sistema, quanto menor o valor (varia de 0 a 99) deste parâmetro maior a importância da tarefa.

- **int* create_attribute(int** *tam_atributo***)**, cria os tipos de dados abstratos chamados atributos, definidos no nosso *framework*, onde estão as variáveis compartilhadas por diversas tarefas no sistema, podendo estas tarefas escrever (*set_attribute*), ler (*get_attribute*), ou realizar ambas as operações sobre estas variáveis. Tem como retorno um apontador para o atributo alocado em memória. A chamada é realizada com a entrada:

  - *tam_atributo* - contém o tamanho do atributo a ser criado, normalmente dado por *sizeof* do tipo do atributo em questão.

- **int get_attribute(int** $*atributo$**)**, retorna o valor do atributo endereçado pelo ponteiro, isto é, realiza a leitura protegida da variável definida em *atributo*;

- **void set_attribute(int** $*atributo$**, int** $dado$**)**, atribui o valor de $dado$ à posição de memória endereçada por *atributo*, isto é, realiza a escrita protegida de um valor em uma variável definida em atributo;

- **FILA* create_queue(int** tam_fila**)**, cria os tipos de dados abstratos chamados filas (assim como os atributos, eles foram definidos no nosso *framework*), que implementa uma fila circular, onde serão armazenados valores que representam a evolução de uma determinada entrada (ou saída) no decorrer da execução das tarefas, podendo estas inserir (*put_queue*) ou retirar (*get_queue*) elementos da fila. Retorna um apontador para o tipo estruturado definido por FILA. Tem como parâmetro de entrada:

  - tam_fila - estipula o tamanho máximo de elementos que a fila irá reter, após estrapolado este tamanho, se continuarem as operações de escrita, os elementos mais antigos serão perdidos.

- **void put_queue(FILA** $*fila$**, int** $dado$**)**, inclui $dado$ no fim da $fila$, ou seja, realiza a operação de inserção de elementos na fila;
- **int get_queue(FILA** $*fila$**)**, retorna o valor do primeiro elemento da fila realizando uma operação de retirada deste elemento da fila.
- **int activate_all()**, esta chamada de sistema insere todas as tarefas na lista de PRONTO, realizando a transição ESPERA-PRONTO, todas as tarefas são ativadas, no tempo inicial. Após isto o kernel assume a responsabilidade sobre a execução das tarefas que são escalonadas de acordo com o período e a prioridade de cada uma.
- **void end_cycle()**, esta primitiva finaliza o ciclo da tarefa inserindo-a na lista de ESPERA se ainda não venceu o seu período, despachando logo em seguida uma nova tarefa para execução.

5.6 Mecanismos internos do μKernel

5.6.1 Escalonamento

O mecanismo de escalonamento utilizado pelo μKernel é realizado através das funções *schedule* (escalonador) e *dispatch* (despachante). A primitiva *schedule* verifica se a tarefa que está executando tem maior prioridade do que as tarefas que estão na lista de pronto esperando. Se for positivo, nenhuma ação é tomada e a tarefa continua a executar. Por outro lado, se a verificação encontrar uma tarefa de maior prioridade esperando, a tarefa que está executando é inserida na lista de pronto e a tarefa de maior prioridade da lista de pronto é despachada para execução. A primitiva *dispatch* retira a tarefa de maior prioridade da lista de pronto e a coloca para executar.

O algoritmo de escalonamento utilizado foi o *Rate Monotonic*[4] (RM) onde as prioridades são atribuídas às tarefas antes da execução e não podem ser modificadas durante ela. O RM é intrinsecamente preemptivo, isto é, a tarefa corrente em execução sofre preempção sempre que uma nova tarefa de maior prioridade (e menor período) for ativada, isto é, colocada na lista de pronto.

5.6.2 Estados das tarefas

Como em qualquer sistema operacional que suporte execução concorrente de múltiplas tarefas em um único processador, o μKernel apresenta três estados principais nos quais as tarefas podem estar, conforme a Figura 5.4:

- **Executando** - Uma tarefa entra neste estado quando inicia, ou continua, sua execução no processador. Neste estado a tarefa tem a posse da CPU;

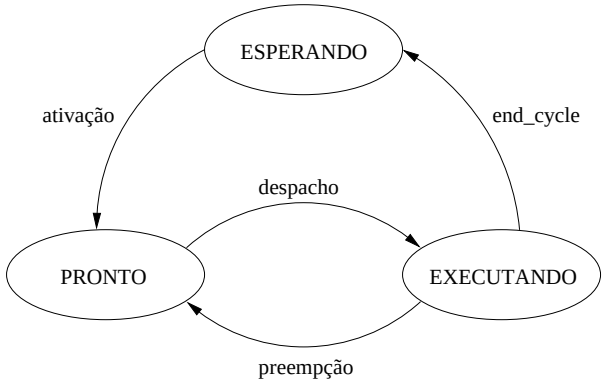

Figura 5.4: Diagrama de Estados das tarefas do μKernel.

- **Pronto** - Este estado é aquele no qual as tarefas estão prontas para executar mas não executam porque o processador está ocupado por outra tarefa. Todas as tarefas que estão nesta condição são mantidas em uma lista chamada lista de pronto (ou *ready_queue*);
- **Esperando** - Uma tarefa entra neste estado quando, após ter tomado posse da CPU, tenha completado sua execução com a primitiva *end_cycle*, neste instante o relógio local da tarefa passa a marcar a hora da próxima ativação e uma nova tarefa da lista de pronto é despachada para execução. Assim como o estado anterior as tarefas que estão neste estado são mantidas em uma lista de espera (ou *sleep_queue*).

Já que as tarefas são escalonadas baseadas no RM (*Rate Monotonic*), todas as listas de estado das tarefas no μKernel são ordenadas por ordem decrescente de prioridade. Fazendo assim a tarefa de maior prioridade pode ser simplesmente extraída do início da lista. Entretanto, uma operação de inserção requer uma varredura sobre diversos elementos da lista. Todas as listas implementadas são simplesmente encadeadas, ou seja, possuem somente um apontador para o próximo elemento. A operação de inserção, realizada pela primitiva *insert_task*, recebe como parâmetro a tarefa a ser inserida e um apontador para a lista na qual esta será inserida.

5.6.3 Manipulação do tempo

Para gerar uma referência de tempo um circuito de *timer* do DSP é programado para interromper o processador em taxas fixas. O intervalo de tempo com o qual o *timer* é programado para interromper define a unidade de tempo no sistema, que é o intervalo de tempo mínimo a ser manipulado pelo μKernel (resolução). A unidade de tempo no sistema também pode ser chamada de *tick*.

O valor assumido pelo *tick* depende da aplicação especificada. Em geral, valores pequenos do *tick* melhoram a resposta do sistema e permitem manipular atividades periódicas com altas taxas de

ativação. Por outro lado, um *tick* muito pequeno causa um grande *overhead* na execução da rotina de manipulação do tempo.

A rotina de manipulação do tempo é realizada pela chamada periódica de duas primitivas (uma após a outra) *wake_up* e *time_stamp*. A primitiva *wake_up* verifica cada tarefa da lista de espera retirando aquelas que vencem o período e inserindo-as na lista de pronto de acordo com a prioridade de execução de cada uma. Já a primitiva *time_stamp* incrementa os tempos de processamento de todas as tarefas que estão na lista de pronto e da tarefa que está executando. Sendo assim, quando a tarefa termina a sua execução (com a primitiva *end_cycle*), é verificado se o tempo decorrido da sua ativação até o término da sua execução excede o seu período. Caso isto aconteça, a tarefa é colocada diretamente na lista de pronto ao invés de ser colocada na lista de espera.

A cada interrupção do *timer* a rotina de manipulação do tempo:

- Salva o contexto da tarefa em execução;
- Remove as tarefas que serão ativadas da lista de espera e insere na lista de pronto conforme a prioridade de cada uma;
- Atualiza o relógio das tarefas, ou seja, incrementando os tempos de processamento;
- Realiza o escalonamento, isto é, testa se existe tarefa de maior prioridade esperando para executar;
- Restaura o contexto da tarefa corrente;
- Retorna da interrupção.

5.6.4 Estruturas de dados utilizadas

Como em qualquer sistema operacional, a informação sobre a tarefa está armazenada em uma estrutura de dados, o descritor da tarefa (TCB ou *Task Control Block*). Em particular, um TCB contém todos os parâmetros especificados pelo programador em tempo de criação da tarefa, mais outras informações necessárias para que o kernel possa gerenciar a tarefa. No μKernel foi criado um TCB com as seguintes informações:

- O endereço de memória correspondente ao início da tarefa;
- O tipo de tarefa (HARD, NRT);
- O período de ativação da tarefa;
- O tempo da tarefa, ou seja, a quantidade de *ticks* decorridos desde sua ativação até o término de seu processamento;

- Um relógio local da tarefa que armazena, quando do término do processamento, o período subtraído do tempo da tarefa que seria um cálculo do tempo a esperar até a próxima ativação;
- A prioridade da tarefa, quanto menor o valor (variando de 0 a 99) maior a prioridade;
- Um vetor que contém o contexto da tarefa, isto é, valor de registradores, pilha de *hardware*, acumulador, contador de programa (PC) e outros. No caso do TMS320C2407A, processador sobre o qual foi implementado o μKernel, os registradores salvos foram:
  - registradores de *status*: ST0 e ST1;
  - acumulador: ACC;
  - contador de programa: PC (armazenado no primeiro nível da pilha de *hardware*);
  - pilha de *hardware* (7 níveis): HWSx (x = 2,3,4,5,6,7,8);
  - registrador de multiplicação: PREG;
  - registrador de multiplicação auxiliar: TREG;
  - registradores auxiliares: ARx (x = 1,2,3,4,5,6,7).
- Um vetor de rascunho, onde são armazenados valores de rascunho durante a execução da tarefa;

Como o TCB é inserido em uma lista encadeada de tarefas a ser manipulada pelo μKernel uma informação adicional é o apontador para o próximo elemento da lista.

Outra estrutura de dados utilizada pelo μKernel é o tipo abstrato, definido no capítulo anterior, chamado FILA, esta estrutura possui as seguintes informações:

- Um apontador para o vetor de dados, já que estes são criados dinamicamente de acordo com a especificação do usuário;
- O início e o fim da FILA para a inserção e remoção de elementos;
- O tamanho da FILA, pois a quantidade de dados que pode ser criada pelo usuário não é especificada pelo μKernel, para que seja circular e se possa controlar a FILA de forma eficiente.

Assim como o tipo abstrato FILA existe outro tipo abstrato chamado ATRIBUTO, que é deve ser representado por uma estrutura de dados definida pelo usuário mas alocada em memória pelo μKernel através da primitiva *create_attribute* que, como foi visto na API, retorna um apontador para o tipo ATRIBUTO criado. A organização dos campos internos dependem da informação que o usuário pretende compartilhar entre as tarefas.

5.6.5 Classes de tarefas

As aplicações de controle, usualmente, consistem de atividades computacionais com diferentes características. Por exemplo, as tarefas podem ser periódicas, aperiódicas, dirigidas pelo tempo ou por eventos e podem ter diferentes níveis de criticidade. Para simplificar o μKernel, de forma a diminuir a sobrecarga de processamento, foram consideradas somente duas classes de tarefas:

- Tarefas HARD, com períodos críticos e prioridades altas, e
- Tarefas não tempo real (NRT), que possuem prioridades as mais baixas possíveis.

As tarefas HARD são ativadas periodicamente e finalizadas pela primitiva *end_cycle* quando então são colocadas na lista de espera até a próxima ativação, ou seja, no próximo período. Além disso, dentro do conjunto de tarefas HARD estas são ordenadas conforme a prioridade de cada uma. Já as tarefas NRT não possuem primitiva de término de finalização até por que possuem uma prioridade tão baixa (o máximo possível) que sua rotina acaba sendo executada em *background* pelo μKernel, ou seja, quando o processador não está ocupado por tarefas HARD.

5.6.6 Chaveamento de contexto

Basicamente são duas primitivas de baixo nível (escritas em linguagem de máquina) do μKernel que implementam um mecanismo para salvamento e restauração do contexto de uma tarefa, que é, armazenar os valores dos registradores do processador (descritos na subseção 5.6.4, sobre o vetor de contexto do TCB). Estas tarefas são realizadas pelas primitivas *save_context* (para salvamento) e *load_context* (para restauração).

5.6.7 Comunicação entre tarefas

A comunicação entre tarefas se utiliza de dois tipos abstratos, FILA e ATRIBUTO, já definidos no *framework* proposto e suas primitivas constam da descrição da API. Internamente ao *kernel* trata-se de um mecanismo que garante a exclusão mútua através da não permissão de ocorrências de interrupção durante a execução das operações de criação (*create_attribute* e *create_queue*), leitura (*get_attribute* e *get_queue*) e escrita (*set_attribute* e *put_queue*). Essa não ocorrência é realizada desabilitando-se as interrupções no início da rotina e reabilitando-as ao final, garantindo-se assim a atomicidade da operação. Cada mecanismo implementa internamente métodos de acesso distintos de acordo com a necessidade de sua especificação.

O mecanismo de FILA implementa uma fila circular FIFO (*First In First Out*) com prioridade sobre a inclusão de elementos, isto quer dizer que os elementos antigos, não retirados, são sobrescritos

por elementos novos. Aqui, além de implementar a retirada (*get_queue*) e a inserção (*put_queue*) protegida de elementos, as operações atuam sobre a FILA alterando apontadores internos para o início e fim desta.

Diferentemente do tipo abstrato FILA, o tipo abstrato ATRIBUTO implementa uma estrutura de dados definida pelo usuário que, usualmente, contém elementos de configuração ou de armazenamento. No ATRIBUTO somente o último valor interessa, isto é, na configuração de um controlador não interessa o valor passado e sim o valor mais recente possível. Aqui realiza-se, somente, a leitura (*get_attribute*) e escrita (*set_attribute*) protegida de variáveis globais compartilhadas.

5.7 Conclusão

Neste capítulo foi descrito o μKernel desenvolvido para dar suporte ao *framework* proposto no Capítulo 4. Abrange a parte de especificação da plataforma alvo e o levantamento dos requisitos e da estrutura necessária ao μKernel. Após a parte de projeto partiu-se para o levantamento da API (serviços do usuário) e então especificou-se seus mecanismos internos.

O código fonte do μKernel tem aproximadamente 1.000 linhas de código, parte escrita em linguagem *C* e parte em linguagem *Assembly* mais uma grande quantidade de comentários, o que não podia ser diferente quando se trabalha com este tipo de codificação. O código fonte do μKernel desenvolvido é de uso público e encontra-se listado no Apêndice A desta dissertação. Alguns dados mais técnicos de utilização da memória do DSP podem ser vistos na Tabela 5.1.

Itens avaliados	**Valores**
Memória de programa: fonte	$\simeq 0,9 Kwords$
Memória de dados estática: globais	$\simeq 20 words$
Memória de dados dinâmica (*heap*): por tarefa	$\simeq 45 words$
Memória de dados dinâmica (*heap*): por FILA	$\simeq 5 words$

Tabela 5.1: Dados de utilização da memória do DSP.

Durante o desenvolvimento do μKernel enfrentou-se alguns problemas com a parte de depuração dos programas devido à plataforma alvo executar em uma velocidade muito superior à depuração via porta paralela do PC e por isso foi de grande valia a utilização de um osciloscópio. Apesar disso a ferramenta utilizada (*Code Composer*) oferece um ambiente integrado suficiente para se poder deduzir o tipo de erro que está ocorrendo. O compilador para a linguagem *C* também parece não funcionar tão bem quanto deveria, o que resultou em algumas adaptações do código (como a manipulação de endereços de retorno em *assembly* devido ao compilador *C* manipulá-los independentemente em uma memória de rascunho local), mas pôde-se assim vivenciar alguns problemas que a indústria costuma enfrentar no desenvolvimento de seus produtos.

Capítulo 6

Aplicação Exemplo

Neste capítulo será especifi cada uma aplicação multitarefa, utilizando o *framework* proposto (Capítulo 4) e o núcleo de suporte (μKernel - Capítulo 5) desenvolvidos. O sistema será então projetado para execução em um DSP da Texas Instruments.

De forma a oferecer parâmetros para uma possível adoção do *framework* proposto, juntamente com seu μKernel, serão apresentados alguns dados empíricos relevantes.

6.1 Introdução

Criar um processo de desenvolvimento (uma metodologia) para a engenharia de *software* de sistemas embutidos é muito importante, haja visto que são sistemas muitas vezes codifi cados em linguagens de baixo nível e de difícil reengenharia.

O *framework* da Texas Instruments juntamente com o *framework* proposto e um núcleo de suporte operacional fazem com que o processo de desenvolvimento possua um conjunto de técnicas e ferramentas que facilitam o desenvolvimento de sistemas multitarefa baseados em DSP.

O processo de construção de sistemas embutidos segue algumas regras comuns também ao processo de desenvolvimento de sistemas convencionais para computadores pessoais como especifi cação (levantamento de requisitos), projeto e codifi cação. No caso de sistemas com requisitos temporais é importante incluir também um comportamento temporal de forma a validar ou não a adoção do *framework*.

A especifi cação do sistema é a primeira etapa e abrange o levantamento de requisitos de usuário e do sistema de forma a se ter uma visão da aplicação, suas características e restrições. A segunda etapa está no projeto da aplicação no caso do *framework* proposto: o mapeamento da aplicação em

blocos, a conexão entre eles e a divisão em tarefas. A codificação no caso do *framework* é gerada automaticamente tendo o programador que somente preencher os blocos se estes já não fizerem parte da biblioteca de componentes desenvolvidos. Por último é importante verificar o comportamento temporal da aplicação e verificar se atende aos requisitos de tempo estipulados.

6.2 Especificação da Aplicação

Alguns dos problemas que surgem durante o processo de engenharia de requisitos são resultantes da falta de uma nítida separação entre os diferentes níveis de descrição (declarações abstratas de alto nível e declarações detalhadas). Em [23] é feito uma distinção utilizando os termos *requisitos do usuário*, para designar os requisitos abstratos de alto nível, e *requisitos do sistema*, para indicar uma descrição mais detalhada do que o sistema deverá fazer. Esta seção segue o modelo apresentado de forma a documentar a especificação em métodos já bastante utilizados.

6.2.1 Requisitos do usuário

A aplicação utilizará um único *DSP* da família C2000 da Texas Instruments (TMS320C2407A), que deverá ser utilizado para executar múltiplas tarefas tais como: controle de motor, interface de entrada de dados (teclado), interface de saída de dados (*display*), recebimento e envio de dados remotos (via RS-232, serial padrão).

O controle do motor deverá ser a tarefa mais importante da aplicação devendo manter a velocidade do motor conforme a referência e utilizando os parâmetros configurados para o PID e para o PWM que serão feitos através da inserção de dados via interface de entrada (teclado) e visualizados via interface de saída (*display*). Esta configuração também poderá ser feita remotamente através do recebimento de dados, enviados via computador de mesa com interface serial. Este último também poderá receber dados amostrados atualizados, enviados remotamente pelo dispositivo.

A Figura 6.1 ilustra um protótipo da aplicação com suas respectivas tarefas.

6.2.2 Requisitos do sistema

Os requisitos do sistema são descrições mais detalhadas dos requisitos do usuário. Eles podem servir como base para a implementação do sistema e, portanto, devem ser uma especificação completa e consistente de todo o sistema. Os requisitos do sistema são, freqüentemente, classificados como *funcionais* (funções que o sistema deve fornecer, como deve reagir a entradas específicas e como deve se comportar em determinadas situações), *não-funcionais* (restrições sobre serviços ou funções oferecidas pelo sistema, por exemplo, restrições de tempo, padrões, etc.) ou requisitos de *domínio* (que se originam do domínio da aplicação e que refletem as características deste).

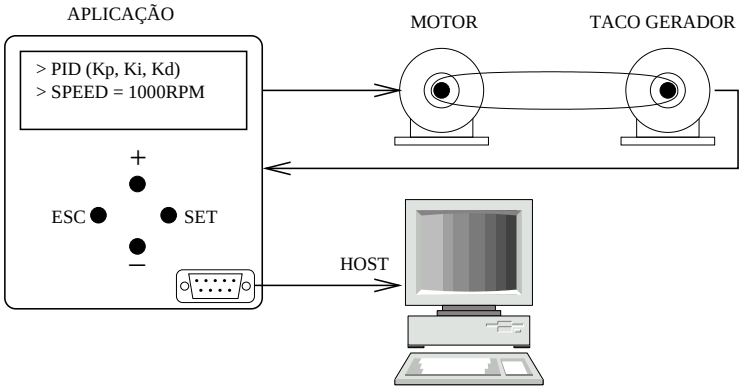

Figura 6.1: Protótipo da aplicação.

6.2.2.1 Requisitos funcionais

Os itens descrevem, detalhadamente, as operações que podem ser realizadas pelo usuário com o objetivo de interagir com a aplicação.

1. O usuário poderá confi gurar os parâmetros do PWM via teclado ou via envio de dados serial;
2. O usuário poderá confi gurar os parâmetros do PID via teclado ou via envio de dados serial;
3. O usuário poderá requisitar amostras de entrada e saída do controlador PID via interface serial.

6.2.2.2 Requisitos não-funcionais

Enquanto a falha em cumprir um requisito funcional individual pode degradar o sistema, a falha em cumprir um requisito não-funcional de sistema pode tornar o sistema inútil. Tendo em vista as restrições impostas pelo desenvolvimento de sistemas embutidos em DSP (tamanho da memória de programa, tamanho da memória de dados, etc.) e as restrições impostas pela aplicação (velocidade de atuação do controlador na malha), chegou-se aos seguintes requisitios não-funcionais:

1. A tarefa de controle deve atuar sobre o motor a cada 50ms (milisegundos);
2. As outras tarefas executam a cada 100ms (milisegundos) e seguem a seguinte ordem de prioridade: teclado, *display*, envio e recebimento de dados remotos;
3. A freqüência de processamento do DSP utilizado é de 20MHz;

6.3 Projeto da aplicação

O projeto da aplicação é a etapa responsável por separar as diversas tarefas e criar, ou reutilizar, os componentes necessários por cada uma. Os elementos que compõem o projeto da aplicação fazem parte dos *frameworks* estudados anteriormente (Capítulos 3 e 4) e serão aplicados no sistema proposto.

Primeiramente são identificadas as tarefas e separadas conforme suas prioridades, criam-se então os blocos que representam os *hardwares* externos e por conseqüência os componentes que realizam a interface entre a tarefa e o *hardware* externo (*driver*). Após isto criam-se os componentes internos a cada tarefa. É necessário também inserir os mecanismos de comunicação entre as tarefas quando estes forem necessários.

Pode-se identificar, a partir da especificação, 5 (cinco) tarefas, onde as prioridades na execução são fixas e decrescentes:

- Uma tarefa para controlar um motor, de alta prioridade e com período de 50ms (milisegundos);
- Uma tarefa para interface de entrada de dados (teclado) com período de 100ms (milisegundos);
- Uma tarefa para interface de saída de dados (*display*) com período de 100ms (milisegundos);
- Uma tarefa para envio de dados via comunicação serial com período de 100ms (milisegundos);
- E uma tarefa para recebimento de dados via comunicação serial com período de 100ms (milisegundos).

Com o intuito de facilitar a visualização a tarefa, será apresentada sob forma de blocos funcionais e com código gerado a partir deles.

6.3.1 Tarefa de controle do motor

A tarefa de controle do motor, principal tarefa do sistema, tem como objetivo levar e manter a velocidade do motor (*fb*) o mais próximo possível da referência (*ref*). O controle é realizado por um controlador PID (Proporcional Integral Derivativo) e exercido sobre o motor através de um PWM (*Pulse Width Modulation* - modulação por largura de pulso). O sistema é realimentado por um ADC (*Analogic to Digital Converter*) que está conectado a um taco gerador.

Os seguintes blocos foram definidos para atender a esta tarefa:

- *Hardware* externo: Motor e Taco Gerador;
- Manipuladores de dispositivos periféricos (*device drivers*): ADC_DRV e PWM_DRV;

- Algoritmo: Controlador PID.

Os parâmetros do PID e do PWM recebem, inicialmente, valores pré-configurados mas podem ser configurados através da entrada de dados. A tarefa de controle (Figura 6.2) funciona com a atuação do PWM sobre o motor que está conectado a uma taco gerador que realimenta a tarefa através do conversor analógico digital (ADC). O controlador verifica a realimentação e conforme os parâmetros configurados recalcula a largura de pulso do PWM.

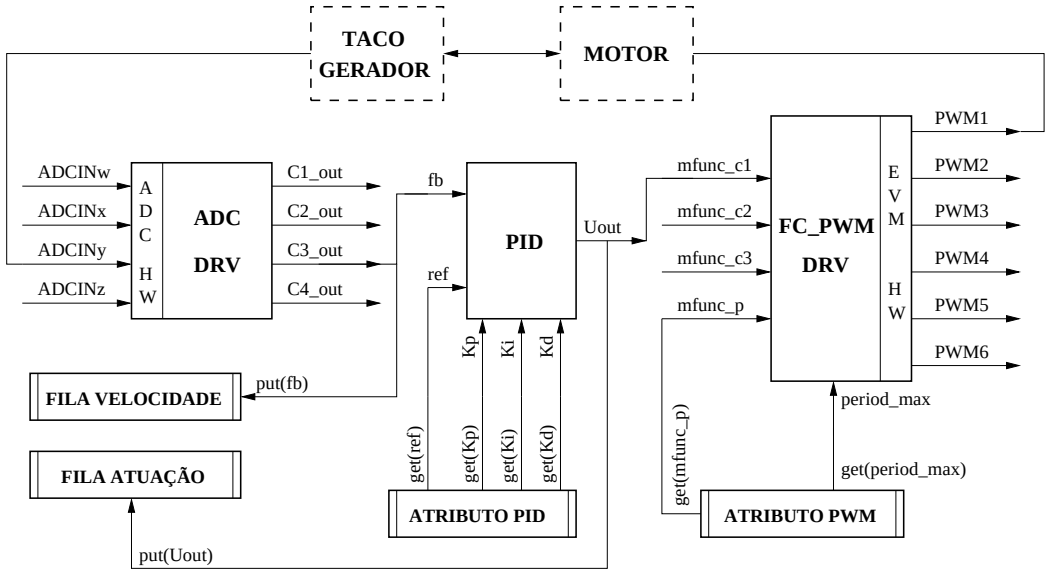

Figure 6.2: Tarefa de controle do motor

Seguindo a metodologia proposta para conexão entre componentes, o Algoritmo 6.1 ilustra o código gerado por esta tarefa.

Algoritmo 6.1: Tarefa de controle do motor

```
void motorControlThread(void)
{
        while(1)
        {
                /* UPDATE THE ADC INPUT */
                adc.update(&adc);

                /* CONFIGURE PID ATTRIBUTES */
                pid.Kp  = get_attribute(PID_Kp);
                pid.Ki  = get_attribute(PID_Ki);
                pid.Kd  = get_attribute(PID_Kd);
                pid.ref = get_attribute(PID_ref);

                /* CONNECT ADC OUTPUT TO PID INPUT */
                pid.fb  = adc.c3_out;
```

```
                /* PUT PID INPUT IN QUEUE */
                put_queue(VELOCIDADE, adc.c3_out);

                /* COMPUTE PID CONTROLLER STRATEGY */
                pid.calc(&pid);

                /* PUT PID OUTPUT IN QUEUE */
                put_queue(ATUACAO, pid.Uout);

                /* CONNECT PID OUTPUT TO PWM INPUT */
                pwm.mfunc_c1 = pid.Uout;

                /* CONFIGURE PWM ATTRIBUTES */
                pwm.mfunc_p    = get_attribute(PWM_mfunc_p);
                pwm.period_max = get_attribute(PWM_period_max);

                /* UPDATE THE PWM OUTPUT */
                pwm.update(&pwm);

                /* END EXECUTION CYCLE */
                end_cycle();
        }
}
```

6.3.2 Tarefa de entrada de dados (teclado)

A função de entrada de dados é confi gurar o sistema através da seleção, incremento e decremento dos parâmetros do controlador PID e do PWM.

A entrada de dados (Figura 6.3) é realizada através de quatro teclas (seleção, retorna, incremento e decremento) que são conectadas à entrada digital (IOP - *Input Output Port*) que fornece, através de seu *driver*, a operação selecionada ao algoritmo de controle do teclado responsável por realizá-la. Blocos identifi cados:

- *Hardware* externo: Interface de Teclado;
- Manipulador de dispositivo periférico (*device driver*): TECLADO_DRV;
- Algoritmo: Controle de teclado.

O código gerado a partir da conexão entre os blocos pode ser visto no Algoritmo 6.2.

Algoritmo 6.2: Tarefa de entrada de dados (teclado)

```
void keypadThread(void)
```

```
{
        while(1)
        {
                /* GET KEY SELECTED BY USER AND SET THE KEYPAD OPTION */
                tc_drv.update(&tc_drv);
                tc.op     = tc_drv.data;

                /* GET DATA FROM ATTRIBUTES */
                tc.estado = get_attribute(ESTADO);
                tc.v1     = get_attribute(PID_Kp);
                tc.v2     = get_attribute(PID_Ki);
                tc.v3     = get_attribute(PID_Kd);
                tc.v4     = get_attribute(PID_ref);
                tc.v5     = get_attribute(PWM_period_max);
                tc.v6     = get_attribute(PWM_mfunc_p);

                /* UPDATE THE KEYPAD STATE MACHINE */
                tc.update(&tc);

                /* SET DATA IN ATTRIBUTES */
                set_attribute(ESTADO,         tc.estado);
                set_attribute(PID_Kp,         tc.v1);
                set_attribute(PID_Ki,         tc.v2);
                set_attribute(PID_Kd,         tc.v3);
                set_attribute(PID_ref,        tc.v4);
                set_attribute(PWM_period_max, tc.v5);
                set_attribute(PWM_mfunc_p,    tc.v6);

                /* END EXECUTION CYCLE */
                end_cycle();
        }
}
```

6.3.3 Tarefa de saída de dados (*display*)

A saída de dados tem por função mostrar o modo atual selecionado, possibilitando que a entrada de dados seja realizada de forma visual.

A tarefa (Figura 6.4) é composta por um algoritmo de controle do *display* que recolhe as informações do PID e do PWM, formata e envia, através do *driver* da porta digital (IOP), para um *hardware* externo composto por um LCD (*Liquid Crystal Display - display* de cristal líquido). Blocos identifi cados:

- *Hardware* externo: Interface de *display*;

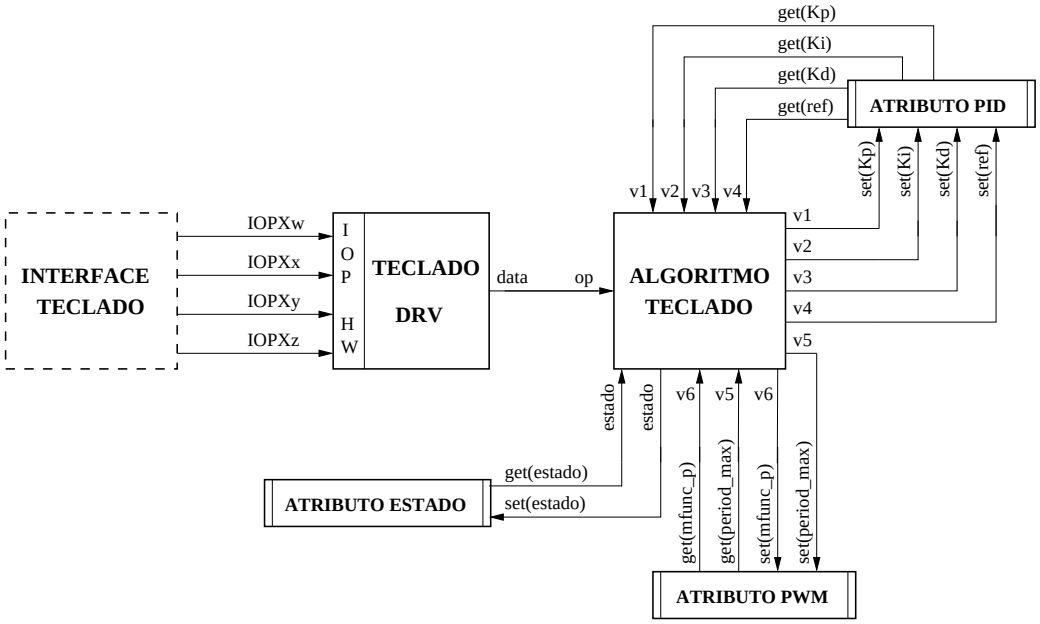

Figure 6.3: Tarefa de entrada de dados (teclado)

- Manipulador de dispositivo periférico (*device driver*): DISPLAY_DRV;
- Algoritmo: Controle de *display*.

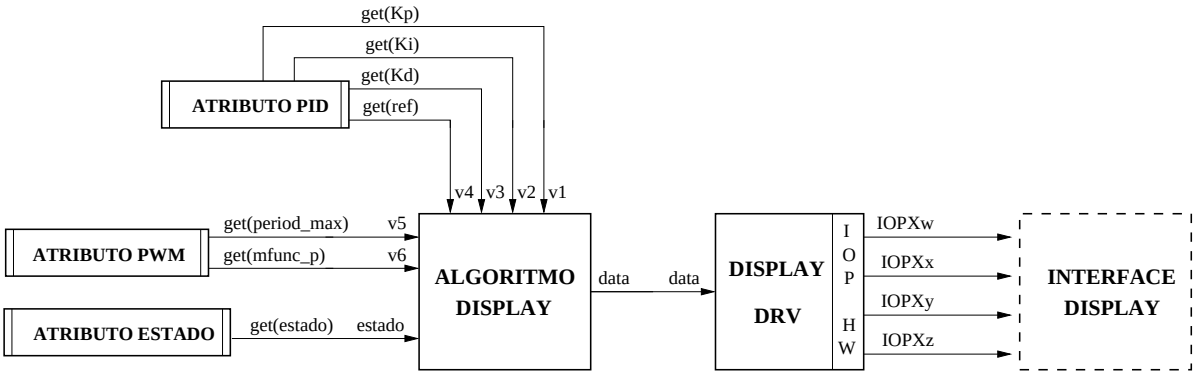

Figure 6.4: Tarefa de saída de dados

O Algoritmo 6.3 ilustra o código gerado pela tarefa.

Algoritmo 6.3: Tarefa de saída de dados (*display*)

```
void displayThread(void)
{
        while(1)
        {
                /* GET DATA FROM ATTRIBUTES */
                dy.estado   = get_attribute(ESTADO);
```

```
                dy.v1       = get_attribute(PID_Kp);
                dy.v2       = get_attribute(PID_Ki);
                dy.v3       = get_attribute(PID_Kd);
                dy.v4       = get_attribute(PID_ref);
                dy.v5       = get_attribute(PWM_period_max);
                dy.v6       = get_attribute(PWM_mfunc_p);

                /* SEND DATA TO DISPLAY */
                dy.update(&dy);
                dy_drv.data = dy.data;
                dy_drv.update(&dy_drv);

                /* END EXECUTION CYCLE */
                end_cycle();
        }
}
```

6.3.4 Acesso Remoto

O acesso remoto é composto por duas tarefas: uma para enviar outra para receber dados utilizando uma comunicação serial com um computador de mesa através de um driver de comunicação serial (SCI - *Serial Communication Interface*), utilizando uma porta padrão RS-232 (serial padrão utilizada em computadores de mesa).

A tarefa de envio de dados (Figura 6.5) é realizada através de um algoritmo próprio que envia os atributos do PID e do PWM, além dos valores amostrados para entrada e saída (atuação) do controlador PID para um programa residente localizado no computador de mesa. Blocos identificados:

- *Hardware* externo: RS-232;
- Manipulador de dispositivo periférico: SCI_TX;
- Algoritmo: Protocolo de envio de dados.

A tarefa de recepção de dados (Figura 6.6) também feita por algoritmo próprio que recebe os atributos do PID e do PWM para configuração dos blocos funcionais, estes atributos são configurados também através do programa residente localizado no computador de mesa. Os blocos identificados são similares aos da tarefa anterior:

- *Hardware* externo: RS-232;
- Manipulador de dispositivo periférico: SCI_RX;

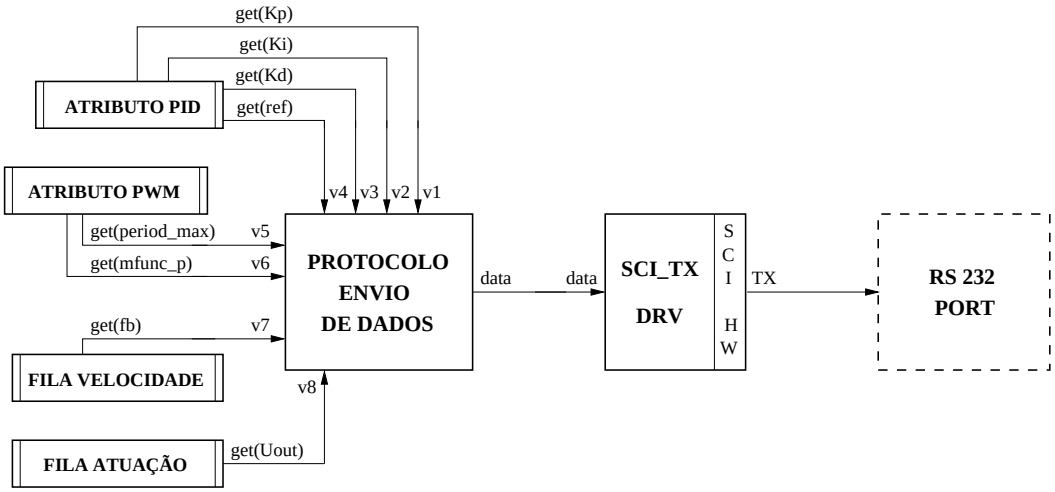

Figure 6.5: Tarefa de envio de dados

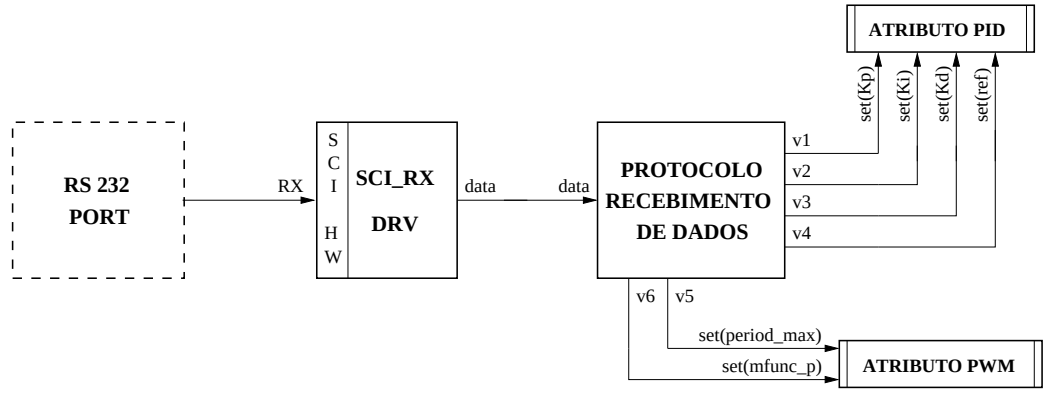

Figure 6.6: Tarefa de recepção de dados

- Algoritmo: Protocolo de recebimento de dados.

O código gerado pelas tarefas segue o modelo proposto e está listado nos Algoritmos 6.4 e 6.5 que representam, respectivamente, as tarefas de envio e de recepção de dados pelo dispositivo.

Algoritmo 6.4: Tarefa de envio de dados

```
void sendRemoteThread(void)
{
        while(1)
        {
                /* GET DATA FROM ATTRIBUTES AND QUEUES */
                tx.v1 = get_attribute(PID_Kp);
                tx.v2 = get_attribute(PID_Ki);
                tx.v3 = get_attribute(PID_Kd);
                tx.v4 = get_attribute(PID_ref);
                tx.v5 = get_attribute(PWM_period_max);
                tx.v6 = get_attribute(PWM_mfunc_p);
```

```
                tx.v7 = get_queue(VELOCIDADE, fb);
                tx.v8 = get_queue(ATUACAO, Uout);

                /* SEND DATA TO SERIAL RECEIVER */
                tx.update(&tx);
                sci_tx.data = tx.data;
                sci_tx.update(&sci_tx);

                /* END EXECUTION CYCLE */
                end_cycle();
        }
}
```

Algoritmo 6.5: Tarefa de recepção de dados

```
void receiveRemoteThread(void)
{
        while(1)
        {
                /* RECEIVE DATA FROM SERIAL SENDER */
                sci_rx.update(&sci_rx);
                rx.data = sci_rx.data;
                rx.update(&rx);

                /* SET RECEIVED DATA IN ATTRIBUTE */
                set_attribute(PID_Kp, rx.v1);
                set_attribute(PID_Ki, rx.v2);
                set_attribute(PID_Kd, rx.v3);
                set_attribute(PID_ref, rx.v4);
                set_attribute(PWM_period_max, rx.v5);
                set_attribute(PWM_mfunc_p, rx.v6);

                /* END EXECUTION CYCLE */
                end_cycle();
        }
}
```

6.3.5 Compondo as várias tarefas

O processo de composição das diversas tarefas tem como objetivo possibilitar ao engenheiro de *software* uma visão da aplicação como um todo. A composição é feita de forma simples envolvendo os limites das tarefas com retângulos tracejados de cantos arredondados (conforme explicado no Capítulo 4) os tipos abstratos são compartilhados entre as tarefas e, portanto, não fazem parte de somente uma tendo que ser acessados

através de primitivas do núcleo de suporte. A aplicação como um todo está ilustrada na Figura 6.7.

O código listado no Algoritmo 6.6 representa a inicialização do sistema com a criação e instanciação de todos os componentes, tipos abstratos e tarefas.

Algoritmo 6.6: Código principal da aplicação

```
/* INCLUDING THE RTOS - REAL-TIME OPERATION SYSTEM */
#include<kernel.h>

/* CREATE HARDWARE DEPENDENT COMPONENTS */
ADC_DRV adc = ADC_DRV_DEFAULTS;
PWM_DRV pwm = PWM_DRV_DEFAULTS;
SCI_RX_DRV sci_rx = SCI_RX_DRV_DEFAULTS;
SCI_TX_DRV sci_tx = SCI_TX_DRV_DEFAULTS;
TECLADO_DRV tc_drv = TECLADO_DRV_DEFAULTS;
DISPLAY_DRV dy_drv = DISPLAY_DRV_DEFAULTS;

/* CREATE THREADS DEPENDENT COMPONENTS */
TECLADO tc = TECLADO_DEFAULTS;
DISPLAY dy = DISPLAY_DEFAULTS;
PROTOCOLO_TX tx = PROTOCOLO_TX_DEFAULTS;
PROTOCOLO_RX rx = PROTOCOLO_RX_DEFAULTS;
PID pid = PID_DEFAULTS;

/* CREATE ATTRIBUTES AND QUEUES */
typedef struct
        {
        int Kp;
        int Ki;
        int Kd;
        int ref;
         } ATRIBUTO_PID;

ATRIBUTO_PID *atr_pid;
#define PID_Kp &atr_pid->Kp
#define PID_Ki &atr_pid->Ki
#define PID_Kd &atr_pid->Kd
#define PID_ref &atr_pid->ref

typedef struct
        {
        int period_max;
        int mfunc_p;
        } ATRIBUTO_PWM;

ATRIBUTO_PWM *atr_pwm;
#define PWM_period_max &atr_pwm->period_max
```

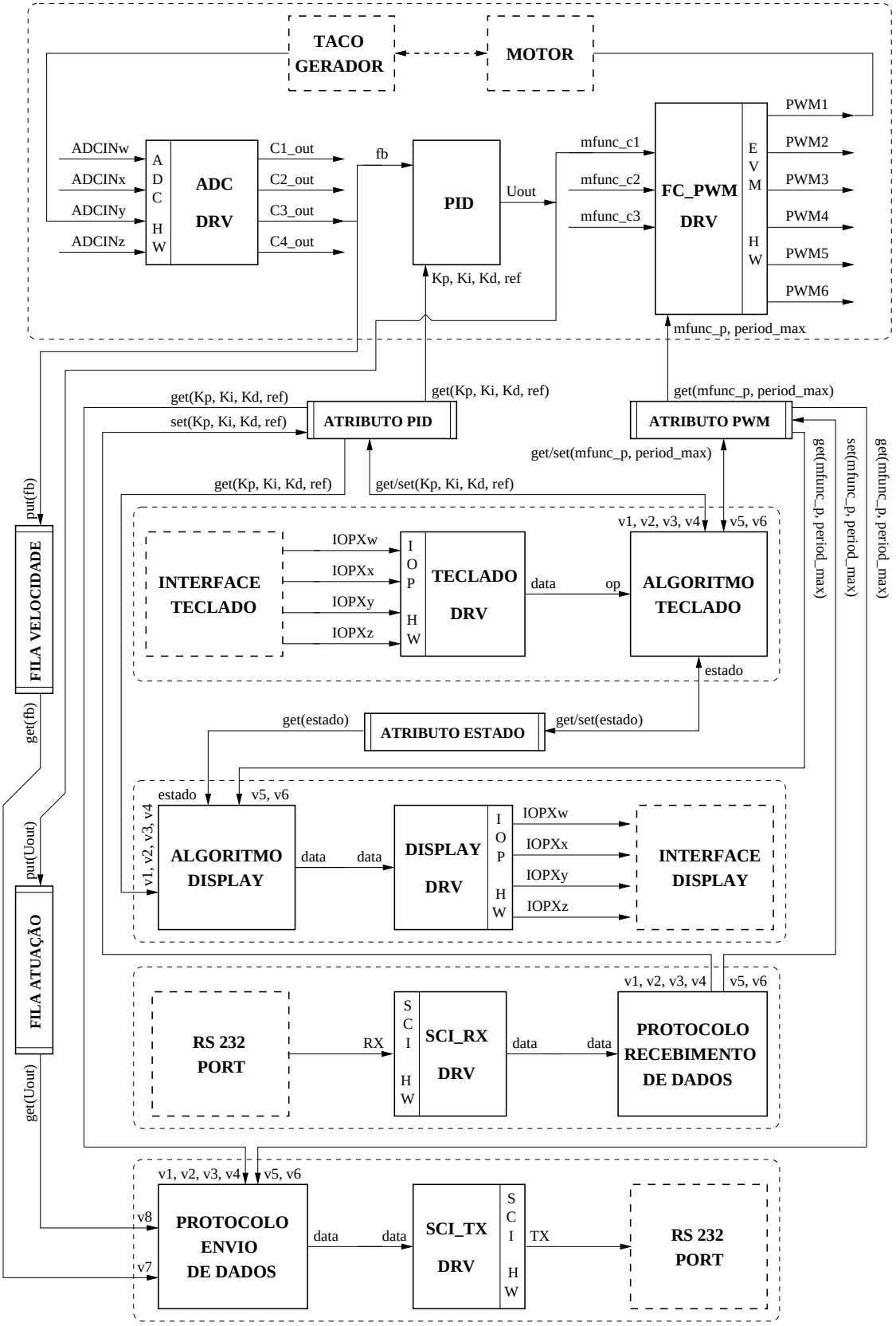

Figura 6.7: Tarefas

```
#define PWM_mfunc_p &atr_pwm->mfunc_p

typedef struct
        {
        int estado;
        } ATRIBUTO_ESTADO;

ATRIBUTO_ESTADO *atr_estado;
#define ESTADO &atr_estado->estado

FILA *fila_velocidade;
#define VELOCIDADE fila_velocidade
FILA *fila_atuacao;
#define ATUACAO fila_atuacao

#define TAM_FILA 10

/* MAIN FUNCTION */
void main(void)
{
        /* KERNEL INITIALIZATION */
        init_system(5 /* ms */);

        /* HARDWARE COMPONENTS INITIALIZATION */
        adc.init(&adc);
        pwm.init(&pwm);
        sci_rx.init(&sci_rx);
        sci_tx.init(&sci_tx);
        dy_drv.init(&dy_drv);
        tc_drv.init(&tc_drv);

        /* THREADS COMPONENTS INITIALIZATION */
        tc.ini(&tc);
        dy.ini(&dy);
        tx.ini(&tx);
        rx.ini(&rx);
        pid.ini(&pid);

        /* INITIALIZE ATTRIBUTES AND QUEUES */
        atr_pid = (ATRIBUTO_PID*)create_attribute(sizeof(ATRIBUTO_PID));
        atr_pwm = (ATRIBUTO_PWM*)create_attribute(sizeof(ATRIBUTO_PWM));
        atr_estado = (ATRIBUTO_ESTADO*)create_attribute(sizeof(ATRIBUTO_ESTADO));
        fila_velocidade = create_queue(TAM_FILA);
        fila_atuacao = create_queue(TAM_FILA);

        /* CREATE TASKS */
        create_task(motorControlThread(),  HARD, 10, 1);
        create_task(keypadThread(),        HARD, 20, 2);
```

```
        create_task(displayThread(),       HARD, 20, 3);
        create_task(sendRemoteThread(),    HARD, 20, 4);
        create_task(receiveRemoteThread(), HARD, 20, 5);

        /* ACTIVATE ALL TASKS */
        active_all();

        /* DO NOTHING */
        while(1);
}
```

6.4 Comportamento temporal

Os aspectos temporais são de grande importância nos sistemas de controle embutidos. De forma complementar ao desenvolvimento do *framework* e do μKernel, foram disponibilizados alguns dados empíricos que exprimem o comportamento temporal da aplicação juntamente com o núcleo de suporte. Como comportamento temporal da aplicação podemos considerar:

- Tempo de computação das tarefas (C_x) - tempo que a tarefa gasta durante seu ciclo de execução;
- Tempo de *overhead* do μKernel (O_x) - o *overhead* é o preço a ser pago pelas facilidades do μKernel, é representado pela soma dos tempos gastos com a execução do núcleo de suporte desde o momento em que a tarefa *x* é inserida na lista de pronto até o término de seu ciclo de execução, e compreende:
  - o chaveamento de contexto entre as tarefas, salvamento dos registradores da tarefa que está deixando o processador e carregamento dos registradores com informações da tarefa que está ganhando o processador;
  - a manipulação das listas de estados, transição das tarefas entre os estados: espera, pronto e executando;
  - o tratador de interrupção do tempo, no caso da aplicação confi gurado para ocorrer a cada 5ms.
- Tempo de resposta das tarefas (R_x) - representado pela soma do tempo de computação da tarefa *x*, do tempo de interferência das tarefas de maior prioridade e dos tempos de *overhead* do μKernel até o término da execução da tarefa *x*;
- Tempo de interferência das tarefas de maior prioridade (I_x) - é a soma dos tempos de computação das tarefas mais prioritárias que a tarefa *x* durante seu tempo de resposta (R_x).

Para o cálculo do tempo de resposta pode-se utilizar:

$$R_x = C_x + I_x + O_x$$

onde I_x é dado por:

$$I_x = \sum_{i=1}^{x-1} \left\lceil \frac{R_x}{P_i} \right\rceil \cdot C_i$$

O tempo de *overhead* do μKernel é equivalente à latência máxima da interrupção, isso é devido ao μKernel executar sempre com as interrupções desabilitadas. Esta escolha simplificou a programação do mesmo, pois não existe concorrência interna no μKernel. Entretanto, perde-se a oportunidade de diminuir este *overhead*. A latência máxima medida para este conjunto de tarefas ficou em, aproximadamente, 400 μs.

Na aplicação proposta foram coletadas amostras de tempo (de forma empírica, através de utilização de um osciloscópio[1]) para cada tarefa e que podem ser vistos na Tabela 6.1. O período de interrupção do tempo foi definido em 5 *ms* ($1/10$ do menor período[2]) e o período e prioridade das tarefas podem ser vistas na Tabela 6.2.

Tarefa (x)	$R_x(ms)$	$C_x(ms)$	$I_x(ms)$	$O_x(ms)$
1	$9,5$	$9,1$	0	$0,4$
2	$17,5$	$7,5$	$9,1$	$0,9$
3	$25,5$	$7,5$	$16,6$	$1,4$
4	$33,5$	$7,5$	$24,1$	$1,9$
5	$42,5$	$7,5$	$31,6$	$3,4$

Table 6.1: Comportamento temporal das tarefas da aplicação.

x	**Tarefa**	**Prioridade**	**Período (P_x)**
1	**motor**	1	50 *ms*
2	**teclado**	2	100 *ms*
3	***display***	3	100 *ms*
4	**envio remoto**	4	100 *ms*
5	**recebimento remoto**	5	100 *ms*

Table 6.2: Período e prioridade das tarefas.

[1] O osciloscópio foi conectado fisicamente á porta digital E do DSP (8 saídas digitais) e cada tarefa colocava o nível de uma saída digital em alto quando iniciava seu ciclo e em baixo quando terminava seu ciclo. Para medir o tempo de computação da tarefa estipulou-se um período grande o suficiente para que nenhuma tarefa interferisse na outra. A latência máxima foi amostrada da mesma forma com a diferença que a saída digital era manipulada na entrada do μKernel para alto e na saída dele para baixo. O restante dos tempo foram calculados segundo as equações apresentadas.

[2] Isso decorre da necessidade de diminuir o atraso causado pela finalização do ciclo de execução de uma tarefa logo após a ocorrência da interrupção do tempo que, quanto maior for, maior será o atraso gerado. É claro, que um tempo de interrupção muito pequeno aumentará muito *overhead* do μKernel e com isso a interferência deste sobre as tarefas.

Pode-se perceber pela análise das tabelas que a tarefa de controle do motor, tarefa de mais alta prioridade na aplicação, sofre somente a interferência do μKernel resultando em um tempo de resposta (R_x) de 9,5 *ms* (resultado da soma de: $C_x = 9,1$, com $O_x = 0,4$). O restante das tarefas, a medida que decrescem as prioridades, vão sofrendo maior interferência por parte das outras tarefas (I_x) e também do μKernel (O_x) resultando em tempos de resposta maiores.

6.5 Conclusão

Este capítulo descreveu a aplicação do *framework* multitarefa proposto no Capítulo 4, juntamente com o μKernel desenvolvido no Capítulo 5, em uma aplicação exemplo que foi executada em um DSP da Texas Instruments.

O objetivo deste foi descrever o processo de desenvolvimento de uma aplicação multitarefa utilizando as ferramentas e metodologias disponibilizadas de forma a criar uma seqüência de passos a serem utilizados no processo de engenharia de *software* deste tipo de sistema.

São apresentados, ao final, algumas métricas de forma a possibilitar uma melhor avaliação do *framework* disponibilizado a partir das quais o desenvolvedor de sistemas embutidos pode se basear numa decisão de adotar ou não o *framework* multitarefa proposto. É importante destacar que não foi objetivo deste trabalho otimizar a execução do μKernel. Logo, existe ainda espaço para tornar o μKernel implementado mais eficiente, o que reduziria o termo O_x relativo aos *overheads*.

.

Capítulo 7

Considerações finais

Este trabalho descreveu uma proposta que visa melhorar o processo de construção (projeto e implementação) de sistemas embutidos multitarefa. Para tanto, o trabalho consistiu, principalmente, da proposta de um *framework* multitarefa e o desenvolvimento de um núcleo operacional de tempo real para dar suporte ao *framework* desenvolvido.

No capítulo 4 foi proposto um *framework* multitarefa alternativo ao modelo monotarefa apresentado no capítulo 3. O modelo monotarefa minimiza muito as funcionalidades de um sistema já que este ficará restrito à execução de uma única tarefa. No caso de sistemas de controle, isso fica restrito a um controlador ou mais, quando todos operam na mesma freqüência. Se o sistema for composto por tarefas que operam em freqüências diferentes o modelo monotarefa não é capaz de atender eficientemente. É sabido que os diagramas de blocos são uma linguagem de modelagem de sistemas muito utilizada entre os engenheiros e mudar este paradigma requer uma reestruturação do conhecimento que este engenheiro agregou durante grande parte de sua formação. Considerando-se isto optou-se por estender o modelo monotarefa, para um modelo multitarefa, sem no entanto perder as potencialidades do projeto utilizando diagramas de blocos.

Um sistema monotarefa é basicamente seqüencial ele executa uma seqüência, de instruções a cada ativação, que normalmente é definida pela ocorrência de uma interrupção programada (*timer*). Em um sistema multitarefa esta execução não acontece mais seqüencialmente já que as várias tarefas agora concorrem pelo processador, ganhando, no caso deste trabalho, aquela de maior prioridade. Para suportar o *framework* multitarefa proposto foi desenvolvido um núcleo operacional, chamado de μKernel, que foi apresentado no capítulo 5 e seu código fonte incluído no apêndice.

Finalmente, no capítulo 6 foi mostrado uma aplicação para exemplificar a utilização do *framework* proposto juntamente com o μKernel desenvolvido. A aplicação desenvolvida teve como objetivo ilustrar o processo de desenvolvimento de um sistema embutido utilizando o diagrama de blocos funcionais sob um paradigma multitarefa além de testar a execução eficiente do núcleo de suporte.

A aplicação desenvolvida teve como base as necessidades reais de um sistema produzido na indústria onde, normalmente, é composto por uma controlador, uma interface humano-máquina e um meio de comunicação com computador de mesa ou algumas vezes rede local. A implementação dos componentes seguiu todo o processo estudado e o processo de projeto e implementação das tarefas e do μKernel teve como foco principal a viabilidade do *framework* proposto.

Espera-se com este trabalho contribuir para a melhoria dos processos de desenvolvimento de *software* para sistemas embutidos no contexto do controle e automação.

O trabalho, a princípio, tinha como objetivo secundário implementar a aplicação em um protótipo real o qual teve o *hardware* fornecido por empresa parceira. Infelizmente, devido a falta de tempo, este objetivo não se concretizou e os componentes que representam os blocos apenas simulam as suas funcionalidades. Esse e outros trabalhos futuros podem ser sugeridos, como:

- Otimização do μKernel - não foi objetivo deste trabalho otimizar a execução das tarefas internas do μKernel e sim prover um núcleo operacional de suporte ao *framework* desenvolvido. O *overhead* gerado por ele, como foi apresentado, está em aproximadamente 400μs para cada interrupção do tempo em que são executados os mecanismos internos do μKernel, o que pode ser restritivo para alguns tipos de aplicações de controle que precisam de um tempo de resposta muito pequeno. Para reduzir o *overhead* produzido pelo núcleo operacional atual que representa quase 6% das execuções das tarefas da aplicação desenvolvida, é necessário otimizar operações realizadas pelo μKernel, como:
  - passar algumas operações feitas em linguagem C para linguagem *Assembly*;
  - melhorar o algoritmo para manipulação das listas de estados ou até mesmo retirá-las mudando de um mecanismo de escalonamento periódico para um mecanismo do tipo *one-shot*, reduzindo o *overhead* significativamente;
  - otimizar o salvamento de contexto, retirando alguns registradores que podem ser descartados para uma aplicação específica.

- Criação de uma ferramenta para programação visual - a programação visual é uma poderosa ferramenta a ser utilizada e a opção de se ter mantido os diagramas de blocos foi especialmente devido à familiaridade que os engenheiro possuem com ele. Durante o trabalho foi preocupação ilustrar e mostrar o código gerado seja por um componente isolado seja pela composição com diversos componentes e/ou tarefas, isso para permitir a implementação de uma ferramenta de programação visual que siga os mesmos padrões adotados no trabalho;

- Criação de uma biblioteca de componentes - com a implementação de uma ferramenta de programação visual nada mais natural que esta possua um repositório onde possam ser guardados os componentes já desenvolvidos para reutilizações em futuros projetos. Um dos objetivos de manter o diagrama de blocos mapeado em diagrama de componentes é a reusabilidade que

pode ser dada na criação dos sistemas, aumentando a efi ciência do desenvolvedor e diminuindo o tempo de liberação de um novo produto para o mercado;

- Emprego do *framework* proposto e do μKernel desenvolvido na construção de uma aplicação real, de preferência por alguma empresa do setor, depois de desenvolvido a ferramenta visual e a biblioteca de componentes citados anteriormente, avaliando métricas como: tempo de desenvolvimento antes e depois; facilidade de utilização; requisitos temporais mantidos; e outras.

Apêndice A

Código fonte do μKernel

O código fonte do μKernel está disponível para uso público (para ter acesso a ele entrar em contato com *marcos@das.ufsc.br*)e dividido em três partes:

- Cabeçalho (*kernel.h*) - inclui o TCB (*Task Control Block*), a definição do tipo abstrato FILA e as assinaturas das primitivas da API (serviços do usuário).
- Código em linguagem *C* (*kernel.c*) - inclui grande parte das implementações das primitivas, tanto da API quanto internas, também possui várias definições utilizadas para a configuração de registradores na inicialização do sistema.
- Código em linguagem *Assembly* (*kernel.asm*) - contém aquelas primitivas que só podem ser escritas em baixo nível pois não podem sofrer alterações de um compilador de alto nível. Estão aqui o salvamento e recuperação do contexto, o tratador de interrupções do tempo, ativação das tarefas e término do ciclo de uma tarefa.

De maneira a tornar o mais legível o código fonte foi divido nas mesmas partes acima citadas.

A.1 Cabeçalho (kernel.h)

Algoritmo A.1: Cabeçalho do μKernel

```
/*
============================================================================
Nome do Arquivo:  kernel.h
Origem:           DAS REAL-TIME GROUP
Descricao:
Este arquivo define as constantes acessiveis para a inicializacao do
```

```
componente, os prototipos para as funcoes implementadas no arquivo
kernel.c e contem a definicao da estrutura do componente.
================================================================================
Historico de versoes:
--------------------------------------------------------------------------------
 Liberado em 20-01-2004 - versao 1.0
--------------------------------------------------------------------------------
*/

#ifndef __kernel_H__
#define __kernel_H__

#ifndef NULL
#define NULL 0
#endif

/*--------------------------------------------------------------------------------
 Define os tipos de tarefas
--------------------------------------------------------------------------------*/
#define HARD  0
#define NRT   1
/*--------------------------------------------------------------------------------
--------------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------------
 Define as estruturas abstratas utilizadas no kernel
/*--------------------------------------------------------------------------------
 FILA - tipo abstrato criado pela chamada do metodo create_queue(tam_fila),
 onde tam_fila representa a quantidade de elementos da fila. Seus elementos
 podem ser acessados pelos metodos put_queue(), que insere um elemento na
 fila e get_queue(), que retira um elemento da fila. Se o tamanho da fila for
 insuficiente para a quantidade de dados inseridos os dados mais antigos vao
 sendo substituidos por dados novos a cada insercao ateh que retiradas sejam
 realizadas, se tratando portanto de uma fila circular.
--------------------------------------------------------------------------------*/
typedef struct
        {
        int *dado;    /* ponteiro para criacao de um vetor de elementos */
        int inicio;   /* posicao onde estah o inicio da fila            */
        int fim;      /* posicao onde estah o fim da fila               */
        int tam_fila; /* tamanho da fila                                */
        } FILA;
/*--------------------------------------------------------------------------------
 TCB - Task Control Block ou Descritor da Tarefa - armazena as informacoes
 principais necessarias para a correta execucao da tarefa. A criacao de uma
 tarefa se dah chamando o metodo create_task(addr,tipo,periodo,prioridade).
 Apos criada, a tarefa eh inserida na fila de espera ateh que seja chamado o
 metodo para ativacao das tarefas onde a responsabilidade de execucao das
 tarefas, de acordo com parametros escolhidos, eh colocada sobre o kernel.
```

```
-------------------------------------------------------------------------*/
#define TAM_CONTEXTO 22
#define TAM_RASCUNHO 15
typedef struct tcb
        {
        void (*addr)(void); /* endereco de execucao da tarefa               */
        int  tipo;          /* tipo da tarefa: PERIODIC, SPORADIC, APERIODIC */
        int  periodo;       /* periodo de ativacao da tarefa                */
        int  tasktime;      /* tempo da tarefa = qtde de ticks qdo executando*/
        int  relogio;       /* relogio local da tarefa = periodo - tc       */
        int  prioridade;    /* prioridade da tarefa                         */
        int  estado;        /* estado da tarefa: SLEEP,READY,RUN            */
        /*-----------------------------------------------------------------
         CONTEXTO: [0]= ST0; [1]= ST1; [2]= ACCL; [3]= ACCH;
                   [NAO]= HWS1(PC local); [4]= HWS2(PC ant); [5]= HWS3;
                   [6]= HWS4; [7] = HWS5; [8]= HWS6; [9]= HWS7; [10]= HWS8;
                   [11]= PREGH; [12]= PREGL; [13]= TREG; [14]= AR0; [15]= AR1;
                   [16]= AR2; [17]= AR3; [18]= AR4; [19]= AR5; [20]= AR6;
                   [21]= AR7;
        -----------------------------------------------------------------*/
        unsigned int contexto[TAM_CONTEXTO];
        /*-----------------------------------------------------------------
         Devido ao compilador C utilizar uma pequena pilha de rascunho, nao
         prevendo o salvamento deste e portanto muitas vezes invadindo o
         espaco do descritor de uma tarefa, foi necessario criar esta pilha
         de rascunho.
        -----------------------------------------------------------------*/
        unsigned int rascunho[TAM_RASCUNHO];
        struct tcb *next; /* apontador para o proximo descritor da lista */
        } TCB; /* Task Control Block - descritor do processo             */

/*-------------------------------------------------------------------------
 Prototipos das funcoes - primitivas publicas do kernel
 -------------------------------------------------------------------------*/
/* inicializacao do sistema */
void  init_system(unsigned int tick);
/* ativacao de todas as tarefas criadas */
void  activate_all(void);
/* finalizacao do ciclo de uma tarefa */
void  end_cycle(void);
/* criacao de tarefas */
void  create_task(void (*addr)(void), int tipo, int periodo, int prioridade);
/* criacao do tipo abstrato ATRIBUTO */
int*  create_attribute(int tam_atributo);
/* leitura (get) e escrita (set) de atributos */
int   get_attribute(int *atributo);
void  set_attribute(int *atributo, int dado);
/* criacao do tipo abstrato FILA */
```

```
FILA* create_queue(int tam_fila);
/* retirada (get) e escrita (put) de dados na FILA */
int   get_queue(FILA *fila);
void  put_queue(FILA *fila, int dado);
/*-------------------------------------------------------------------------
 -------------------------------------------------------------------------*/
/* FIM DO KERNEL.h */
#endif     /*__kernel_H__ */
```

A.2 Código em linguagem *C* (kernel.c)

Algoritmo A.2: Código em linguagem *C* do µKernel

```
/*
===========================================================================
 Nome do Arquivo:  kernel.c
 Origem:           DAS REAL-TIME GROUP
 Descricao:
 Este arquivo contem a implementacao dos metodos definidos pela interface
 kernel.h para o microKernel de tempo real utilizado no desenvolvimento
 rapido de sistemas orientados a componentes para DSP.
===========================================================================
 Historico de versoes:
---------------------------------------------------------------------------
  Liberado em 25-01-2004 - versao 1.0
---------------------------------------------------------------------------
-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------
 Bibliotecas necessarias para execucao do Kernel e configuraco do sistema
-------------------------------------------------------------------------*/
#include"kernel.h"
#include"regs2407.h"
/*-------------------------------------------------------------------------
-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------
 Variaveis externas - para que o main do programa principal possa ser
 associado a uma tarefa esporadica do sistema que executa quando sobra tempo.
-------------------------------------------------------------------------*/
extern void main(void);
/*-------------------------------------------------------------------------
-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------
 Define os estados das tarefas
-------------------------------------------------------------------------*/
#define READY  1
#define EXE    2
```

```
#define SLEEP  3
/*-------------------------------------------------------------------------
-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------
 Defines para os valores dos registradores para o contexto inicial
/*-------------------------------------------------------------------------
 ST0 - registrador de status #0 (16 bits)
-------------------------------------------------------------------------*/
#define bARP   1              /* bits 15-13 - Auxiliary Register Pointer */
#define bOV    0              /* bit     12 - Overflow Flag */
#define bOVM   0              /* bit     11 - Overflow Mode */
/* RESERVADOS */          /* bits  10-9 - Rsvd(1) e INTM(1) */
#define bDP    4          /* bits   8-0 - Data Page Pointer */
#define rST0 (bARP<<13)+(bOV<<12)+(bOVM<<11)+(3<<9)+(bDP)
/*-------------------------------------------------------------------------
 ST1 - registrador de status #1 (16 bits)
-------------------------------------------------------------------------*/
#define bARB   1       /* Auxiliary Register Pointer Buffer 1 : (AR1) */
#define bCNF   0       /* bit     12 - On-Chip DARAM Configuration */
#define bTC    0       /* bit     11 - Test/Control Flag */
#define bSXM   1       /* bit     10 - Sign-Extension Mode */
#define bC     1       /* bit      9 - Carry */
/* RESERVADOS */       /* bits   8-5 - Rsvd(1111) = 15   */
#define bXF    1       /* bit      4 - XF hardware pin */
/* RESERVADOS */       /* bits   3-2 - Rsvd(11) = 3   */
#define bPM    0       /* bits   1-0 - Product Shift Mode */
#define rST1 (bARB<<13)+(bCNF<<12)+(bTC<<11)+(bSXM<<10)+(bC<<9)+(15<<5)+ \
             (bXF<<4)+(3<<2)+(bPM)
/*-------------------------------------------------------------------------
 ACC - registrador acumulador (32 bits)
-------------------------------------------------------------------------*/
#define rACCH  0       /* registrador Acumulador parte alta  */
#define rACCL  0       /* registrador Acumulador parte baixa */
/*-------------------------------------------------------------------------
 HSW1 - pilha de hardware - 8 niveis (16 bits)
-------------------------------------------------------------------------*/
#define rAR0   0        /* contador de programa - PC */
#define rAR1   0        /* sem valor definido */
#define rPC    0        /* sem valor definido */
#define rHWS3  0        /* sem valor definido */
#define rHWS4  0        /* sem valor definido */
#define rHWS5  0        /* sem valor definido */
#define rHWS6  0        /* sem valor definido */
#define rHWS7  0        /* sem valor definido */
#define rHWS8  0        /* sem valor definido */
/*-------------------------------------------------------------------------
 PREG - registrador de produto (32 bits)
-------------------------------------------------------------------------*/
```

```
#define rPREGH  0        /* registrador de Produto parte alta  */
#define rPREGL  0        /* registrador de Produto parte baixa */
/*-----------------------------------------------------------------------------
 TREG - registrador de multiplicacao (16 bits)
-----------------------------------------------------------------------------*/
#define rTREG   0        /* registrador de Produto parte alta  */
/*-----------------------------------------------------------------------------
 ARn - registrador auxiliares (16 bits)
-----------------------------------------------------------------------------*/
#define rAR2  0        /* registrador Auxiliar 2 */
#define rAR3  0        /* registrador Auxiliar 3 */
#define rAR4  0        /* registrador Auxiliar 4 */
#define rAR5  0        /* registrador Auxiliar 5 */
#define rAR6  0        /* registrador Auxiliar 6 */
#define rAR7  0        /* registrador Auxiliar 7 */
/*-----------------------------------------------------------------------------
 Contexto default para incializacao
-----------------------------------------------------------------------------*/
#define CTXT_DEFAULT {rST1,rST0,rACCL,rACCH,rPC,rHWS3,rHWS4,rHWS5,rHWS6 \
                     ,rHWS7,rHWS8,rPREGH,rPREGL,rTREG,rAR2,rAR3,rAR4,rAR5  \
                     ,rAR6,rAR7}
/*-----------------------------------------------------------------------------
-----------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------
 Defines para os valores dos registradores para configuracao do sistema
/*-----------------------------------------------------------------------------
 SCSR1 - Registrador de controle e status do sistema
-----------------------------------------------------------------------------*/
#define CLKSRC                  0       /* 0 : intern(20MHz)                */
#define LPM                     0       /* 0 : Low power mode 0 if idle     */
#define CLK_PS                  1       /* 001 : PLL multiply by 2          */
#define ADC_CLKEN               0       /* 0 : Nao ADC-service in this test */
#define SCI_CLKEN               0       /* 0 : Nao SCI-service in this test */
#define SPI_CLKEN               0       /* 0 : Nao SPI-servide in this test */
#define CAN_CLKEN               0       /* 0 : Nao CAN-service in this test */
#define EVB_CLKEN               1       /* 1 : Sim EVB-Service - kernel tick */
#define EVA_CLKEN               0       /* 0 : Nao EVA-Service in this test */
#define ILLADR                  1       /* 1 : Clear ILLADR during startup  */
/*-----------------------------------------------------------------------------
 WDCR - Registrador de controle do 'Watchdog'
-----------------------------------------------------------------------------*/
#define WDDIS                   1       /* 0 : Watchdog enabled 1: disabled */
#define WDCHK2                  1       /* 0 : System reset   1: Normal OP  */
#define WDCHK1                  0       /* 0 : Normal Oper.   1: sys reset  */
#define WDCHK0                  1       /* 0 : System reset   1: Normal OP  */
#define WDSP                    7       /* Watchdog prescaler 7 : div 64    */
/*-----------------------------------------------------------------------------
 WSGR - Registrador gerador de estados de espera.
```

```
 Utilizado para comunicacao entre o DSP e perifericos de IO, memoria de dados
 e de programa externa, quando estes possuem resposta lenta (spru357b,pp3-16).
-------------------------------------------------------------------------------*/
#define BVIS                    0       /* 10-9 : 00 Bus visibility OFF      */
#define ISWS                    0       /* 8 -6 : 000 0 Waitstates for IO    */
#define DSWS                    0       /* 5 -3 : 000 0 Waitstates data      */
#define PSWS                    0       /* 2 -0 : 000 0 Waitstates code      */
/*-------------------------------------------------------------------------------
 IFR - Registrador de 'pendencia' da interrupcao da CPU
 Para limpar (0) o bit do IFR deve ser escrito 1 e nao 0 (spru357b,pp2-24).
-------------------------------------------------------------------------------*/
#define fINT6                   0       /* 0 : nao pendente   1: pendente */
#define fINT5                   0       /* 0 : nao pendente   1: pendente */
#define fINT4                   0       /* 0 : nao pendente   1: pendente */
#define fINT3                   1       /* 0 : nao pendente   1: Timer4   */
#define fINT2                   0       /* 0 : nao pendente   1: pendente */
#define fINT1                   0       /* 0 : nao pendente   1: pendente */
/*-------------------------------------------------------------------------------
 IMR - Registrador 'mascarador' de interrupcao da CPU
 Quando uma interrupcao estah mascarada (0) ela nao eh reconhecida independente
 do valor do bit INTM. Quando uma interrupcao estah desmascarada (1) ela eh
 reconhecida se o correspondente bit no IFR estiver em 1 (pendente) e o bit
 INTM estiver em 0 (interrupcoes habilitadas) (spru357b,pp2-25).
-------------------------------------------------------------------------------*/
#define mINT6                   0       /* 0 : mascarada   1: desmascarada */
#define mINT5                   0       /* 0 : mascarada   1: desmascarada */
#define mINT4                   0       /* 0 : mascarada   1: desmascarada */
#define mINT3                   1       /* 0 : mascarada   1: Timer4       */
#define mINT2                   0       /* 0 : mascarada   1: desmascarada */
#define mINT1                   0       /* 0 : mascarada   1: desmascarada */
/*-------------------------------------------------------------------------------
 EVBIFRB - Registrador de 'pendencia' da interrupcao da EVB grupo B
 Para limpar (0) o bit do EVBIFRB deve ser escrito 1 e nao 0 (spru357b,pp6-93).
 Se ocorrerem as condicoes para a interrupcao do periferico (timer4), o
 respectivo bit eh modificado para 1. Uma vez modificado este bit deve ser
 explicitamente 'limpo' or software. Eh obrigatorio fazer isso ou futuras
 interrupcoes nao irao ser reconhecidas. (spru357b,pp6-9).
-------------------------------------------------------------------------------*/
#define fT4OFINT                0       /* timer 4 overflow interrupt  */
#define fT4UFINT                0       /* timer 4 underflow interrupt */
#define fT4CINT                 0       /* timer 4 compare interrupt   */
#define fT4PINT                 1       /* timer 4 period interrupt    */
/*-------------------------------------------------------------------------------
 EVBIMRB - Registrador 'mascarador' de interrupcao da EVB grupo B
 Utilizado para habilitar ou desabilitar as interrupcoes ocorridas no Event
 Manager B (EVB) grupo B. 0 : desabilita 1 : habilita (spru357b,pp6-96).
 As interrupcoes do Event Manager podems ser individualmente habilitadas
 ou desabilitadas pelos registradores 'mascaradores' de interrupcao
```

```
 (spru357b,pp6-9).
-------------------------------------------------------------------------*/
#define eT4OFINT                0       /* timer 4 overflow interrupt  */
#define eT4UFINT                0       /* timer 4 underflow interrupt */
#define eT4CINT                 0       /* timer 4 compare interrupt   */
#define eT4PINT                 1       /* timer 4 period interrupt    */
/*-------------------------------------------------------------------------
-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------
 Defines para os valores dos registradores para configuracao do timer 4
 utilizado pelo kernel para gerar as interrupcoes que manuseiam o
 escalonamento do sistema
/*-------------------------------------------------------------------------
 T4CON - Registrador de controle do timer 4
 PERIODO = (PREESCALER*UPDOWN*AJUSTE)/F_CPU, onde:
         -> PREESCALER = 128;
         -> UPDOWN     = 2;
         -> F_CPU      = 20 MHz;
 portanto, para um PERIODO = 1 ms eh necessario um AJUSTE = 78, aprox.
-------------------------------------------------------------------------*/
#define AJUSTE          78      /* Ajuste necessario para equiv. em ms       */
#define FREE_SOFT        0      /* 00 : Stop immediately on emulation suspend*/
#define TMODE1_0         1      /* 01 : Continuous-Up/Down Count Mode        */
#define TPS2_0           7      /* Clock Preescaler - 111 : CPU clock/128    */
#define T4SWT3           0      /* Timer4 inicia com Timer3 - 0 : desabilita */
#define TENABLE          1      /* Timer Enable - 1 : habilita               */
#define TCLKS1_0         0      /* Clock Source Select - 00 : Internal       */
#define TCLD1_0          0      /* Timer Compare Condition - 00 : Disable    */
#define TECMPR           0      /* Timer compare - 0 : Disable operation     */
#define SELT3PR          0      /* Utiliza Timer3 - 0 : Disable operation    */
/*-------------------------------------------------------------------------
 GPTCONB - Registrador de proposito geral dos timer 3/4
-------------------------------------------------------------------------*/
#define T4STAT           0      /* 0: Counting downward   1: Counting upward */
#define T3STAT           0      /* 0: Counting downward   1: Counting upward */
#define T4TOADC          0      /* 00: No event starts ADC                   */
#define T3TOADC          0      /* 00: No event starts ADC                   */
#define TCOMPOE          0      /* 0: Disable al GP timer compare outputs    */
#define T4PIN            0      /* Polarity of compare output- 00: Forced low*/
#define T3PIN            0      /* Polarity of compare output- 00: Forced low*/
/*-------------------------------------------------------------------------
-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------
 Variaveis globais
-------------------------------------------------------------------------*/
TCB *sleep_queue; /* fila de espera */
TCB *ready_queue; /* fila de pronto */
TCB *exe_task;    /* tarefa executando */
```

```
/* apontador para uma pilha de lixo local, durante a execucao das funcoes
   do kernel. */
unsigned int *lixo;
/*---------------------------------------------------------------------------
 Variaveis globais utilizadas para o salvamento e carregamento do contexto
---------------------------------------------------------------------------*/
/* ponteiro para a pilha do contexto */
unsigned int *ppc;
/* armazena o valor do registrador AR1 antes de apontar para a pilha */
unsigned int ar1;
/* armazena o valor do primeiro endereco de retorno da pilha */
unsigned int end_ret;
/*---------------------------------------------------------------------------
 Prototipos das funcoes locais - primitivas protegidas do Kernel
 ---------------------------------------------------------------------------*/
void  wake_up(void);
void  schedule(void);
void  dispatch(void);
void  sleep_task(void);
void  time_stamp(void);
TCB*  insert_task(TCB *task, TCB *queue);
/*---------------------------------------------------------------------------
---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------
 Funcao de incializacao do mKernel
 Descricao: configura a unidade de tempo (tick) da interrupcao do timer e
            habilita a interrupcao deste para cada tick, foi reservado o
            Timer4 (T4) para gerar o periodo das interrupcoes do mKernel.
---------------------------------------------------------------------------*/
void init_system(unsigned int tick /* ms */)
{
        /* desabilita interrupcoes da CPU */
        asm("\t DINT");

        /* configurando o gerador de estados de espera */
        WSGR=((BVIS<<9)+(ISWS<<6)+(DSWS<<3)+PSWS);

        /* configurando o 'Watchdog timer' para operacao normal */
        WDCR=((WDDIS<<6)+(WDCHK2<<5)+(WDCHK1<<4)+(WDCHK0<<3)+WDSP);

        /* configurando os registradores para reconhecimento da interrupcao
         * do timer pela CPU. */
        SCSR1= ((CLKSRC<<14)+(LPM<<12)+(CLK_PS<<9)+(ADC_CLKEN<<7)+
                   (SCI_CLKEN<<6)+(SPI_CLKEN<<5)+(CAN_CLKEN<<4)+
                   (EVB_CLKEN<<3)+(EVA_CLKEN<<2)+ILLADR);
        IFR = ((fINT6<<5)+(fINT5<<4)+(fINT4<<3)+(fINT3<<2)+(fINT2<<1)+fINT1);
        IMR = ((mINT6<<5)+(mINT5<<4)+(mINT4<<3)+(mINT3<<2)+(mINT2<<1)+mINT1);
```

```
        /* configurando os registradores para reconhecimento da interrupcao
         * do timer pelo Event Manager B (EVB) grupo B. O reset da EVBIFRB
         * deve ser repetido dentro da rotina de tratamento da interrupcao
         * sob pena de nao ser mais receber o reconhecimento da interrupcao
         * pela CPU (spru357b,pp6-9). */
        EVBIFRB = ((fT4OFINT<<3)+(fT4UFINT<<2)+(fT4CINT<<1)+fT4PINT);
        EVBIMRB = ((eT4OFINT<<3)+(eT4UFINT<<2)+(eT4CINT<<1)+eT4PINT);

        /* configurando o Timer4 que serah utlizado como ciclo de tempo para
         * a execucao das funcoes do kernel e despacho de uma nova tarefa. */
        GPTCONB = ((T4STAT<<14)+(T3STAT<<13)+(T4TOADC<<9)+(T3TOADC<<7)+
                  (TCOMPOE<<6)+(T4PIN<<2)+T3PIN);
        T4CNT   = NULL; /* incializando o contador do Timer4 com 0 */
        T4PR    = AJUSTE*tick;/* periodo do Timer4 */
        T4CON   = ((FREE_SOFT<<14)+(TMODE1_0<<11)+(TPS2_0<<8)+(T4SWT3<<7)+
                  (TENABLE<<6)+(TCLKS1_0<<4)+(TCLD1_0<<2)+(TECMPR<<1)+SELT3PR);

        /* inicializa as filas de estado */
        exe_task    = NULL;
        sleep_queue = NULL;
        ready_queue = NULL;

        /* criando o lixo que serah a memoria de rascunho local do uKernel */
        lixo = (unsigned int*)malloc(10*sizeof(unsigned int));

        /* cria o TCB da tarefa NRT main e despacha para execucao */
        create_task(main, NRT, 99, 99);
        /* coloca a tarefa main na fila de PRONTO */
        wake_up();
        /* despacha a tarefa NRT main para execucao */
        dispatch();
        /* inicialmente ppc deve apontar para o inicio da 1a. pilha
           o restante serah administrado pelo salvamento ou restauracao
           do contexto */
        ppc = exe_task->contexto;
}

/*-----------------------------------------------------------------------------
 Acorda as tarefas da fila de ESPERA.
 Descricao: Verifica se existem tarefas na fila de ESPERA.
            Percorre a fila de ESPERA testando cada tarefa, se a tarefa
            possuir seu relogio igual 0, esta na hora de acordah-la, i.e.,
            insere a tarefa na fila de PRONTO, senao decrementa o relogio da
            tarefa e a mantem na fila de ESPERA.
            Neste metodo tambem eh reativado o flag para reconhecimento das
            interrupcoes da EVB onde estah localizado o timer utilizado pelo
            uKernel para manuseamento do tempo.
-----------------------------------------------------------------------------*/
```

```
void wake_up(void)
{
        TCB *task = sleep_queue;
        TCB *aux;
        sleep_queue = NULL;

        /* enquanto exisitrem tarefas insere-as nas respectivas fila de acordo
         * com o relogio local que marca a hora para a proxima ativacao. */
        while(task != NULL)
        {
                aux = task->next;

                if(task->relogio == NULL)
                        ready_queue = insert_task(task, ready_queue);
                else
                {
                        task->relogio--;
                        sleep_queue = insert_task(task, sleep_queue);
                }

                task = aux;
        }

        /* configurando os registradores para reconhecimento da interrupcao
         * do timer pelo Event Manager B (EVB) grupo B. Deve ser repetido
         * dentro da rotina de tratamento da interrupcao (spru357b,pp6-9). */
        EVBIFRB = ((fT4OFINT<<3)+(fT4UFINT<<2)+(fT4CINT<<1)+fT4PINT);
}

/*---------------------------------------------------------------------------
 Acerta o tempo de computacao das tarefas.
 Descricao: Verifica se existem tarefas na fila de PRONTO e para todas as
            tarefas periodicas incrementa o relogio de cada uma.
            Tambem incrementa o tempo da tarefa que estah executando caso
            esta tambem seja periodica.
---------------------------------------------------------------------------*/
void time_stamp(void)
{
        TCB *task = ready_queue;
        while(task != NULL)
        {
                if(task->tipo == HARD)
                        task->tasktime++;
                task = task->next;
        }
        /* atualizando o tempo de processamento da tarefa que estah executando */
        if(exe_task != NULL)
                if((exe_task->tipo) == HARD)
```

```
                        exe_task->tasktime = (exe_task->tasktime)+1;

}
/*---------------------------------------------------------------------------
 Coloca a tarefa na fila de ESPERA.
 Descricao: Terminado o ciclo de execucao da tarefa, insere a tarefa que estava
            executando na fila de ESPERA (SLEEP).
            Se o periodo da tarefa menos o tempo de processamento desta
            resultarem em 0 ou um valor < que 0, insere a tarefa na fila de
            PRONTO ao inves de inserir na fila de ESPERA.
---------------------------------------------------------------------------*/
void sleep_task(void)
{
        /* testando se jah nao estourou o periodo da tarefa */
        if( ((exe_task->periodo)-(exe_task->tasktime)) <= 0 )
        {
                exe_task->estado   = READY;
                ready_queue = insert_task(exe_task, ready_queue);
        }
        else
        {
                exe_task->estado   = SLEEP;
                exe_task->relogio  = (exe_task->periodo)-(exe_task->tasktime);
                sleep_queue = insert_task(exe_task, sleep_queue);
        }
        exe_task->tasktime = NULL;
        exe_task = NULL;
}

/*---------------------------------------------------------------------------
 Escalonamento das tarefas.
 Descricao: Seleciona a tarefa com maior prioridade para ganhar o processador.
            Se a tarefa de maior prioridade for a que estiver esperando na
            fila de PRONTO, entao insere a que estava executando na fila de
            PRONTO e despacha a tarefa de maior prioridade para execucao.
---------------------------------------------------------------------------*/
void schedule(void)
{
        if((exe_task != NULL)&&(ready_queue != NULL))
                if(ready_queue->prioridade < exe_task->prioridade)
                {
                        exe_task->estado = READY;
                        ready_queue = insert_task(exe_task, ready_queue);
                        dispatch();
                }
}

/*---------------------------------------------------------------------------
```

```
 Despacha a tarefa para execucao.
 Descricao: Entrega para a CPU a primeira tarefa da fila de PRONTO, isto e,
            a tarefa de maior prioridade para ser executada.
-------------------------------------------------------------------------*/
void dispatch(void)
{
        exe_task         = ready_queue;
        ready_queue      = ready_queue->next;
        exe_task->next   = NULL;
        exe_task->estado = EXE;
        ppc              = &exe_task->contexto[TAM_CONTEXTO-1];
}

/*-------------------------------------------------------------------------
 Cria a tarefa e a coloca na fila de ESPERA
 Descricao: a primitiva create_task aloca e inicializa todas as estruturas
            necessarias por uma tarefa e a coloca na fila de ESPERA (SLEEP).
-------------------------------------------------------------------------*/
void create_task(void (*addr)(void), int tipo, int periodo, int prioridade)
{
        int i;
        TCB *task;
        unsigned int ctxt_default[TAM_CONTEXTO] = CTXT_DEFAULT;

        /* desabilita interrupcoes da CPU */
        asm("\t DINT");

        task = (TCB*)malloc(sizeof(TCB));
        task->addr       = addr;
        task->tipo       = tipo;
        task->periodo    = periodo;
        task->tasktime   = NULL;
        task->relogio    = NULL;
        task->prioridade = prioridade;
        task->estado     = SLEEP;
        task->next       = NULL;
        /* inicializando a pilha do contexto */
        for(i=0; i<TAM_CONTEXTO; i++)
                task->contexto[i] = ctxt_default[i];
        /* inicializando a pilha de rascunho */
        for(i=0; i<TAM_RASCUNHO; i++)
                task->rascunho[i] = NULL;

        task->contexto[4]  = (unsigned int)task->addr;
        task->contexto[15] = (unsigned int)task->rascunho;
        /*-----------------------------------------------------------------
         CONTEXTO: [0]= ST0; [1]= ST1; [2]= ACCL; [3]= ACCH;
                   [NAO]= HWS1(PC local); [4]= HWS2(PC ant); [5]= HWS3;
```

```
                    [6]= HWS4; [7] = HWS5; [8]= HWS6; [9]= HWS7; [10]= HWS8;
                    [11]= PREGH; [12]= PREGL; [13]= TREG; [14]= AR0; [15]= AR1;
                    [16]= AR2; [17]= AR3; [18]= AR4; [19]= AR5; [20]= AR6;
                    [21]= AR7;
        ------------------------------------------------------------------------*/

        /* Insere a tarefa na fila de ESPERA (sleep_queue) */
        sleep_queue = insert_task(task, sleep_queue);

        /* habilita interrupcoes da CPU */
        asm("\t EINT");
}

/*-------------------------------------------------------------------------------
 Insere uma tarefa na fila de estados selecionada: SLEEP ou READY.
 Descricao: Percorre a fila de tarefas (queue) passada como parametro e insere
            a tarefa (task) antes de uma tarefa de menor prioridade que ela.
            Retorna a fila de tarefas remanejada. O intuito de ter retorno eh
            que se a fila nao contiver elementos, i.e. igual NULL, entao esta
            deve ser retornada apontada para 'task'.
-------------------------------------------------------------------------------*/
TCB* insert_task(TCB *task, TCB *queue)
{
        TCB *ant  = NULL;
        TCB *prox = queue;

        /* posiciona para insercao de 'task' antes da tarefa de menor
         * prioridade que 'task' */
        while((prox != NULL) && (task->prioridade >= prox->prioridade))
        {
                ant = prox;
                prox = prox->next;
        }

        /* insere a tarefa 'task' na posicao crescente de prioridade */
        if(ant != NULL)
                ant->next = task;
        else
                queue = task;

        task->next = prox;

        return queue;
}

/*-------------------------------------------------------------------------------
 Funcao de criacao do tipo abstrato denominado ATRIBUTO
 Descricao: Este tipo abstrato se refere a uma estrutura de dados criada pelo
```

```
            usuario e alocada por esta funcao com o objetivo de permitir uma
            comunicacao segura entre tarefas no sistema, sem que haja problemas
            de acesso simultaneo aos diversos atributos. O principio estah na
            funcao receber o tamanho do atributo (um sizeof da estrutura
            criada pelo usuario) e retornar este tipo abstrato instanciado.
            Pode ser acessado por dois metodos: uma funcao para 'setar' o valor
            de um dado do ATRIBUTO (set_attribute(...)) e outra funcao para
            'ler' o valor de uma dado do ATRIBUTO (get_attribute(...)).
-------------------------------------------------------------------------*/
int* create_attribute(int tam_atributo)
{
        int *p = NULL;

        /* desabilita interrupcoes da CPU */
        asm("\t DINT");

        /* instanciando o ATRIBUTO */
        p = (int*)malloc(tam_atributo);

        /* habilita interrupcoes da CPU */
        asm("\t EINT");

        return p;
}

/*-------------------------------------------------------------------------
 Funcao que retorna um determinado dado de um ATRIBUTO
 Descricao: Recebe o endereco do dado a ser 'lido' do ATRIBUTO, acessando-o
            de forma protegida e retornando uma copia do dado apos a leitura.
-------------------------------------------------------------------------*/
int get_attribute(int *atributo)
{
        int dado = 0;

        /* desabilita interrupcoes da CPU */
        asm("\t DINT");

        /* copiando o valor do dado do ATRIBUTO para ser retornado */
        dado = *atributo;

        /* habilita interrupcoes da CPU */
        asm("\t EINT");

        return dado;
}

/*-------------------------------------------------------------------------
 Funcao que atribui o valor de dado a um determinado ATRIBUTO
```

```
 Descricao: Recebe o endereco do dado no ATRIBUTO e o valor a ser colocado no
            lugar, realizando a operacao de atribuicao de forma protegida.
-------------------------------------------------------------------------------*/
void set_attribute(int *atributo, int dado)
{
        /* desabilita interrupcoes da CPU */
        asm("\t DINT");

        /* atribuindo o valor de dado ao dado do ATRIBUTO */
        *atributo = dado;

        /* habilita interrupcoes da CPU */
        asm("\t EINT");
}

/*-------------------------------------------------------------------------------
 Funcao de criacao do tipo abstrato denominado FILA
 Descricao: Este tipo abstrato se refere a uma fila circular de tamanho
            variavel a ser escolhido pelo usuario no momento de criacao da
            FILA, para isto possui em sua estrutura um atributo que referencia
            o seu tamanho (tam_fila). Possui metodos de acesso atraves de duas
            funcoes, uma para insercao de elementos (put_queue(...)) e outra
            para retirada de elementos (get_queue(...)).
-------------------------------------------------------------------------------*/
FILA* create_queue(int tam_fila)
{
        int i;
        FILA *p = NULL;

        /* desabilita interrupcoes da CPU */
        asm("\t DINT");

        /* inicializando o tipo abstrato FILA */
        p = (FILA*)malloc(sizeof(FILA));
        p->dado = (int*)malloc(tam_fila*sizeof(int));
        p->tam_fila = tam_fila;
        p->inicio = p->fim = tam_fila-1;
        for(i=0;i<tam_fila;i++)
                p->dado[i] = NULL;

        /* habilita interrupcoes da CPU */
        asm("\t EINT");

        return p;
}

/*-------------------------------------------------------------------------------
 Funcao que retorna um elemento de uma determinada FILA, retirando-o desta
```

```
 Descricao: Por se tratar de uma fila circular em que o fim eh flutuante a
            retirada de elemetos da fila acontece ateh que esta chegue ao
            'fim' virtual, se isto ocorrer serah retornado sempre o ultimo
            dado colocado na fila ateh que seja inserido um novo dado.
-------------------------------------------------------------------------------*/
int get_queue(FILA *fila)
{
        int dado = 0;

        /* desabilita interrupcoes da CPU */
        asm("\t DINT");

        /* Se a fila nao estiver vazia posiciona o indice no dado a ser
         * retirado. Senao retorna o ultimo dado ateh que exista novo
         * dado na fila */
        if(fila->inicio != fila->fim)
                fila->inicio =  (fila->inicio == (fila->tam_fila-1))?
                                0 : (fila->inicio)+1;

        /* retira o dado mais antigo da fila */
        dado = fila->dado[fila->inicio];

        /* habilita interrupcoes da CPU */
        asm("\t EINT");

        return dado;
}

/*-------------------------------------------------------------------------------
 Funcao insere um dado em uma determinada FILA
 Descricao: Em se tratando de uma fila circular implementada sobre vetor se
            torna necessario verificar a ocorrencia de estouro (chegou no fim
            do vetor) na insercao de um dado se isto ocorrer este dado devera
            ser inserido no inicio. Para evitar que dados recentes fossem
            perdidos optou-se por descartar os dados mais antigos
            sobreescrevendo-os pelos mais recentes.
-------------------------------------------------------------------------------*/
void put_queue(FILA *fila, int dado)
{
        /* desabilita interrupcoes da CPU */
        asm("\t DINT");

        /* abre espaco para um novo dado posicionando o indice */
        fila->fim = (fila->fim == (fila->tam_fila-1))? 0 : (fila->fim)+1;

        /* verifica a ocorrencia de estouro. Se sim, sobreescreve o dado
         * antigo pelo novo dado, avancando o inicio. */
        if(fila->inicio == fila->fim)
```

```
                fila->inicio =  (fila->inicio == (fila->tam_fila-1))?
                                0 : (fila->inicio)+1;

        /* insere o dado na fila */
        fila->dado[fila->fim] = dado;

        /* habilita interrupcoes da CPU */
        asm("\t EINT");
}

/*-------------------------------------------------------------------------
 Rotina de tratamento para as interrupcoes que nao estao sendo utilizadas.
 Descricao: Utilizada para manusear interrupcoes ou eventos nao esperados.
            Todas as interrupcoes nao utilizadas apontam para esta rotina
            a partir do vetor de interrupcoes, assim se qualquer interrupcao
            nao tratada acontecer serah tratada de um forma benigna,
            prevenindo saltos, chamadas ou execucoes nao intencionais.
            Importante notar que esta eh uma rotina de tratamento de
            interrupcoes, nao uma rotina qualquer.
-------------------------------------------------------------------------*/
void interrupt phantom(void)
{
        static int phantom_count;
        phantom_count ++;
}

/*-------------------------------------------------------------------------
 Fim da implementacao da parte C do Microkernel. Arquivo: kernel.c
-------------------------------------------------------------------------*/
```

A.3 Código em linguagem *Assembly* (kernel.asm)

Algoritmo A.3: Código em linguagem *Assembly* do μKernel

```
;=========================================================================
; Nome do Arquivo:  kernel.asm
; Origem:           DAS REAL-TIME GROUP
; Descricao:
; Este arquivo contem a implementacao dos metodos assembly definidos pela
; interface kernel.h para o microKernel de tempo real utilizado no
; desenvolvimento rapido de sistemas orientados a componentes para DSP.
;=========================================================================
; Historico de versoes:
;-------------------------------------------------------------------------
;  Liberado em 23-01-2004 - versao 1.0
;-------------------------------------------------------------------------
```

```
;-------------------------------------------------------------------------
; Diretivas necessarias para a execucao da parte assembly do Kernel:
; .ref - referencias a funcoes declaradas no arquivo kernel.c
; .global - referencias as variaveis globais declaradas no arquivo kernel.c
; .def - declaracao das funcoes escritas assembly e utilizadas pelo usuario,
;        pelo vetor de interrupcoes e internamente.
;-------------------------------------------------------------------------
        .ref _dispatch, _schedule
        .ref _wake_up, _sleep_task, _time_stamp
        .global _ar1, _ppc, _end_ret, _lixo
        .def _activate_all, _timer_handling, _end_cycle
        .def _save_context, _load_context

;-------------------------------------------------------------------------
; Inicio do programa
;-------------------------------------------------------------------------
        .text
; -------------------------------------------------------------------------
; Ativa todas as tarefas
; Descricao: Salva o contexto de main. Insere todas as tarefas na fila de
;            PRONTO. Despacha a tarefa mais prioritaria para execucao.
;            Carrega o contexto da tarefa a executar.
;            Esta funcao foi implementada em Assembly devido ao compilador C
;            inserir codigo para salvar o endereco de retorno, fazendo com que
;            este fosse retirado da pilha de hardware, impossibilitando assim
;            retorno para a tarefa que ira executar depois de carregado o
;            contexto.
; -------------------------------------------------------------------------
_activate_all:
        ; desabilitando a ocorrencia de interrupcoes da CPU
        DINT
        ; salva o contexto da tarefa que tem a CPU
        CALL _save_context
        ; pilha de lixo para o kernel escrever enquanto estiver executando suas
        ; funcoes internas.
        MAR  *,AR1
        LAR  AR1,_lixo
        ; acorda as tarefas que tiverem completado seu periodo
        CALL _wake_up
        ; realiza o escalonamento entre a tarefa de maior prioridade da fila de
        ; PRONTO e a tarefa que estah executando
        CALL _schedule
        ; carrega o contexto da tarefa que receberah a CPU
        CALL _load_context
        ; habilitando a ocorrencia de interrupcoes da CPU
        EINT
        RET
```

```
; ------------------------------------------------------------------------
; Rotina de manipulacao da interrupcao do timer
; Descricao: Interrompe a execucao da tarefa que tem a CPU a cada tick do tempo
;            para acordar (SLEEP -> READY) as tarefas que jah atingiram o seu
;            periodo e envelhece o tempo de computacao das tarefas que jah
;            estao esperando e da tarefa que estah executando. Escalona as
;            tarefas para permitir que a de maior prioridade que estiver na
;            fila de PRONTO possa executar.
; ------------------------------------------------------------------------
_timer_handling:
        ; desabilitando a ocorrencia de interrupcoes da CPU
        DINT
        ; salva o contexto da tarefa que tem a CPU
        CALL _save_context
        ; pilha de lixo para o kernel escrever enquanto estiver executando suas
        ; funcoes internas.
        MAR  *,AR1
        LAR  AR1,_lixo
        ; acorda as tarefas que tiverem completado seu periodo
        CALL _wake_up
        ; atualiza o tempo de processamento das tarefas
        CALL _time_stamp
        ; realiza o escalonamento entre a tarefa de maior prioridade da fila de
        ; PRONTO e a tarefa que estah executando
        CALL _schedule
        ; carrega o contexto da tarefa que receberah a CPU
        CALL _load_context
        ; habilitando a ocorrencia de interrupcoes da CPU
        EINT
        RET

; ------------------------------------------------------------------------
; Termina a execucao da tarefa.
; Descricao: Salva o contexto da tarefa que estava executando.
;            Insere a tarefa que estava executando na fila de ESPERA (SLEEP).
;            Despacha uma nova tarefa para executar.
;            Carrega o contexto da nova tarefa.
; ------------------------------------------------------------------------
_end_cycle:
        ; desabilitando a ocorrencia de interrupcoes da CPU
        DINT
        ; salva o contexto da tarefa que tem a CPU
        CALL _save_context
        ; pilha de lixo para o kernel escrever enquanto estiver executando suas
        ; funcoes internas.
        MAR  *,AR1
        LAR  AR1,_lixo
```

```
        ; coloca a tarefa que tinha a CPU para dormir
        CALL _sleep_task
        ; despacha a tarefa de maior prioridade da fila de pronto para executar
        CALL _dispatch
        ; carrega o contexto da tarefa que receberah a CPU
        CALL _load_context
        ; habilitando a ocorrencia de interrupcoes da CPU
        EINT
        RET

; ---------------------------------------------------------------------------
; Salva o contexto da tarefa em execucao
; Descricao:
; ---------------------------------------------------------------------------
_save_context:
        ; assegurando que o ARP tem como apontador para a pilha o
        ; registrador AR1
        MAR  *,AR1
        ; salvando o registrador auxiliar AR1 pois serah utilizado como
        ; apontador para a pilha do contexto e AR0, ambos utilizados pelo
        ; compilador C. Salvando tbm o endereco de retorno para quem chamou
        ; a funcao de salvamento do contexto (retira da pilha de hw)
        SAR  AR1,_ar1
        ; salvando o endereco de retorno para a funcao que chamou
        POPD _end_ret
        ; utlizando o AR1 como apontador para a pilha do contexto
        LAR  AR1,_ppc
        ; salvando os registradores de status ST0 e ST1
        SST1 *+
        SST  *+
        ; salvando o acumulador (ACC)
        SACL *+
        SACH *+
        ; salvando os 7 niveis da pilha de hardware. Um nivel estah sendo
        ; utilizado para manter o endereco de retorno para quem efetuou o CALL
        POPD *+ ; PC (Program Counter) de quem foi interrompido
        POPD *+ ; end. de retorno qquer
        POPD *+ ; end. de retorno qquer
        POPD *+ ; end. de retorno qquer
        POPD *+ ; end. de retorno qquer
        POPD *+ ; end. de retorno qquer
        POPD *+ ; end. de retorno qquer
        PSHD _end_ret ; restaurando o endereco de retorno de para quem chamou
        ; salvando o valor do registrador PREG
        SPM  0
        SPH  *+
        SPL  *+
        ; salvando o valor do registrador TREG
```

```
        MPYK 1
        SPL  *+
        ; salvando registradores auxiliares (ARn)
        SAR  AR0,*+
        LACL _ar1
        SACL *+
        SAR  AR2,*+
        SAR  AR3,*+
        SAR  AR4,*+
        SAR  AR5,*+
        SAR  AR6,*+
        SAR  AR7,*  ; !!!! lembrar que eu tirei o incremento !!!!
        ; salvando o apontador para o fim da pilha de contexto
        SAR  AR1,_ppc
        ; restaurando o valor original de AR1, AR0 e do endereco de retorno
        LAR  AR1,_ar1
        ; retorna para a funcao que chamou
        RET

; ---------------------------------------------------------------------------
; Carrega o contexto da tareda a ser executada
; Descricao:
; ---------------------------------------------------------------------------
_load_context:
        ; assegurando que o ARP tem como apontador para a pilha o
        ; registrador AR1
        MAR  *,AR1
        ; salvando o endereco de retorno para a funcao que chamou
        POPD _end_ret
        ; utlizando o AR1 como apontador para a pilha do contexto
        LAR  AR1,_ppc
        ; restaurando registradores auxiliares (ARn)
        LAR  AR7,*-
        LAR  AR6,*-
        LAR  AR5,*-
        LAR  AR4,*-
        LAR  AR3,*-
        LAR  AR2,*-
        LACL *-
        SACL _ar1 ; endereco da pilha de rascunho
        LAR  AR0,*-
        ; restaurando o valor do registrador PREG
        LACL *-
        LT   *-
        MPYK 1
        LPH  *
        ; salvando o valor do registrador TREG
        SACL *
```

```
        LT   *-
        ; restaurando os 7 niveis da pilha de hardware. Um nivel estah sendo
        ; utilizado para manter o endereco de retorno para quem efetuou o CALL
        PSHD *- ; end. de retorno qquer
        PSHD *- ; end. de retorno qquer
        PSHD *- ; end. de retorno qquer
        PSHD *- ; end. de retorno qquer
        PSHD *- ; end. de retorno qquer
        PSHD *- ; end. de retorno qquer
        PSHD *- ; PC (Program Counter) de quem foi interrompido
        PSHD _end_ret ; restaurando o endereco de retorno
        ; restaurando o acumulador (ACC)
        LACC *-,16
        ADDS *-
        ; restaurando os registradores de status ST0 e ST1
        LST  *-
        MAR  *,AR1 ; necessario pois LST altera o ARP
        LST1 *     ; !!!! lembrar que eu tirei o decremento !!!!
        MAR  *,AR1 ; necessario pois LST altera o ARP
        ; salvando o apontador para o inicio da pilha de contexto
        SAR  AR1,_ppc
        ; restaurando o valor original de AR1 que aponta para a
        ; pilha de rascunho
        LAR  AR1,_ar1
        ; retorna para a funcao que chamou
        RET
; ------------------------------------------------------------------------------
; Fim do programa
; ------------------------------------------------------------------------------
        .end
```

Referências Bibliográficas

[1] A. S. Berger. *Embedded Systems Design: An Introduction to Process, Tools and Techniques*. CMP Books, 2002.

[2] S. Blonstein. *Reference Frameworks for eXpressDSP Software: A White Paper*. Technical Document - Application Report. Texas Instruments, SDS, December 2002. revision A.

[3] S. Blonstein. *The TMS320 DSP Algorithm Standard*. Technical Document - White Paper. Texas Instruments, May 2002. revision C.

[4] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. The Kluwer international series in engineering and computer science. Real-time systems. Kluwer Academic Publishers, 4 edition, 2002.

[5] A. Cervin. *Towards the Integration of Control and Real-Time Scheduling Design*. Doctor thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, May 2000.

[6] G. DeMicheli. *Hardware / software co-design: applications domains and design technologies*, pages 1–28. Hardware / Software Co-design. Kluwer Academic Publishers, 1996.

[7] A. B. H. Ferreira. *Dicionário Aurélio Eletrônico: Século XXI*. Nova Fronteira, 3.0 edition, Novembro 1999.

[8] D. Figoli. *A Software Modularity Strategy for Digital Control Systems*. Technical Document - Application Report. Texas Instruments, DCSA, March 2001.

[9] W. S. Gan, Y. K. Chong, W. Gong, and W. T. Tan. Rapid prototyping system for teaching real-time digital signal processing. *IEEE Transactions on Education*, 43(1):19–24, 2000.

[10] F. A. C. Gomide and M. L. A. Netto. *Introdução a Automação Industrial Informatizada*. Kapelusz, EBAI (Escola Brasileiro-Argentina de Informática), preliminar edition, 1986.

[11] Digital Control Systems Group, editor. *Digital Motor Control - Software Library*. Technical Document - User Guide. Texas Instruments, DCS Group, August 2001.

[12] Texas Instruments, editor. *TMS320F/C24x DSP Controllers: CPU and Instruction Set*. Technical Document - Reference Guide. Texas Instruments, DSP Solutions, June 1999. revision C.

[13] Texas Instruments, editor. *TMS320 DSP Algorithm Standard - Rules and Guidelines*. Technical Document - User Guide. Texas Instruments, October 2002. revision E.

[14] M. Jamshidi and C. J. Herget. *Computer-Aided Control Systems Engineering*. North-Holland Elsevier Science Publishers, Amsterdam, 1985.

[15] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Kluwer Academic Publishers, 1997.

[16] Q. Li and C. Yao. *Real-time concepts for embedded systems*. CMP Books, 2003.

[17] R. S. Oliveira, A. S. Carissimi, and S. S. Toscani. *Sistemas Operacionais*. Livros didáticos, número 11. Sagra Luzzatto, Instituto de Informática da UFRGS, Porto Alegre, 2 edition, 2001.

[18] U. Rembold, B. O. Nnaji, and A. Storr. *Computer Integrated Manufacturing and Engineering*. Addison-Wesley, 1 edition, 1993.

[19] A. B. Saifuddin. *Computing Environments for Control Engineering*. Doctor of philosophy thesis, Department of Engineering, University of Cambridge, Cambridge, UK, 1996.

[20] J. L. P. Santos and R. S. Oliveira. Aspectos relevantes e confi guração da rede tipo barramento de campo padrão worldfi p aplicada ao sistema de supervisão e controle de uma planta geradora de energia elétrica. In *Anais do XIV Congresso Brasileiro de Automática*, pages 709–714, Natal, RN, 2002.

[21] P. R. Silveira and W. E. Santos. *Automação e Controle Discreto*. Érica, 3 edition, 1998.

[22] D. E. Simon. *An Embedded Software Primer*. Addison-Wesley, 1999.

[23] I. Sommerville. *Engenharia de Software*. Addison-Wesley, 6 edition, 2003. tradução André Maurício de Andrade.

[24] J. Stankovic and K. Ramamrithan, editors. *Tutorial on Hard Real-Time Systems*. IEEE Computer Society Press, 1988.

[25] J. A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10), October 1988.

[26] A. Thé, D. W. Dart, and S. Dirksen. *How to get started with the DSP/BIOS Kernel*. Technical Document - Application Report. Texas Instruments, SFS/SDS, August 2001.

[27] C. A. Valderrama, M. E. De Lima, S. Cavalcante, and E. Barros. *Hardware / Software Co-design: Projetando Hardware e Software Concorrentemente*. Escola de Computação, IME-USP, São Paulo, 2000.

[28] H. Yiu. *Understanding Basic DSP/BIOS Features*. Technical Document - Application Report. Texas Instruments, China, April 2000.